



Universität des Saarlandes

# Analysis of String Sorting using Heapsort

Masterarbeit im Fach Informatik  
Master's Thesis in Computer Science  
von / by

Igor Stassiy

angefertigt unter der Leitung von / supervised by

Prof. Dr. Raimund Seidel

begutachtet von / reviewer

Dr. Victor Alvarez

August 2014



**Hilfsmittelerklärung**

Hiermit versichere ich, die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt zu haben.

**Non-plagiarism Statement**

Hereby I confirm that this thesis is my own work and that I have documented all sources used.

Saarbrücken, den 03. März 2014,

(Igor Stassiy)

**Einverständniserklärung**

Ich bin damit einverstanden, dass meine (bestandene) Arbeit in beiden Versionen in die Bibliothek der Informatik aufgenommen und damit veröffentlicht wird.

**Declaration of Consent**

Herewith I agree that my thesis will be made available through the library of the Computer Science Department at Saarland University.

Saarbrücken, den 03. März 2014,

(Igor Stassiy)

*To my grandparents*

- Igor



# *Abstract*

In this master thesis we analyze the complexity of sorting a set of strings. It was shown that the complexity of sorting strings can be naturally expressed in terms of the prefix trie induced by the set of strings. The model of computation takes into account symbol comparisons and not just comparisons between the strings. The analysis of upper and lower bounds for some classical algorithms such as Quicksort and Mergesort in terms of such a model was shown. Here we extend the analysis to another classical algorithm - Heapsort. We also give analysis for the version of the algorithm that uses Binomial heaps as a heap implementation.



# *Acknowledgements*

I would like to thank my supervisor Prof. Dr. Raimund Seidel for suggesting the question researched in this thesis and for numerous meetings that helped me move forward with the topic. With his expert guidance, more often than not, our discussions ended in a fruitful step forwards towards completing the results to be presented in this thesis.

I would also like to thank Thatchaphol Saranurak for useful discussions on Binomial heapsort and for his valuable suggestions. My gratitude also goes to Dr. Victor Alvarez for agreeing to review this thesis.



# Contents

<b>Abstract</b>	<b>vi</b>
<b>Acknowledgements</b>	<b>viii</b>
<b>1 Introduction</b>	<b>1</b>
1.1 Previous work . . . . .	1
1.2 Problem statement . . . . .	4
1.3 Overview . . . . .	4
<b>2 Heapsort analysis</b>	<b>5</b>
2.1 Overview . . . . .	5
2.2 Randomization . . . . .	8
2.3 Building the heap . . . . .	9
2.4 Different model of building heap . . . . .	15
2.5 Original way of building heap . . . . .	19
<b>3 Binomial Heapsort analysis</b>	<b>31</b>
3.1 Randomness preservation . . . . .	33
3.2 Number of comparisons . . . . .	36
<b>4 Experiments</b>	<b>43</b>
4.1 Binary heapsort . . . . .	43
4.2 Binomial heapsort . . . . .	43
<b>5 Conclusion</b>	<b>45</b>
<b>A Appendix</b>	<b>47</b>
<b>Bibliography</b>	<b>49</b>



# Chapter 1

## Introduction

Sorting is arguably one of the most common routines used both in daily life and in computer science. Efficiency or computation complexity of this procedure is usually measured in terms of the number of “key” comparisons made by the algorithm.

While mainly the problem of sorting is considered solved, up to date there are publications improving the efficiency of sorting when used on real-world data or when used in a different computational model than the standard “key comparison” model [HT02, Han02].

It is well known that the best one can achieve in the “key comparison” model of computation is  $O(n \log(n))$  comparisons when sorting  $n$  keys (more precisely the lower bound is  $\log_2(n!)$ ). However when the keys to be sorted are strings, this complexity measure doesn’t adequately reflect the running time of the sorting algorithm. The reason for this is that comparing two strings lexicographically takes time proportional to the length of their longest common prefix plus one and the model of counting key comparisons simply doesn’t take this fact into account.

A more appropriate model of counting comparisons is to count the total number of symbol comparisons made by the algorithm. It is however not clear what fraction of time will long keys be compared, hence taking a long time, and how many comparisons will involve short keys, hence taking only short time.

### 1.1 Previous work

The core of the problem boils down to choosing parameters of the input strings that well describe the running time of the algorithm on this input instance [Sei10]. If we choose only  $n$  - the number of strings, then by adding a large common prefix  $P$  to all of the

strings, we can force a large number of symbol comparisons, as comparing two strings would require at least  $|P| + 1$  operations. If chosen just  $m$  - the total length of all the strings, adding a common suffix  $S$  to all the strings the number of symbol comparisons stays the same, while  $m$  increases by  $nS$

The problem of parametrization of sorting was dealt with in two ways. The first was developing specialized algorithms for sorting strings. Some of these algorithms perform  $O(m + n \log(n))$  comparisons and can be argued to be worst-case optimal [BS97]. The other approach was to assume that the input strings are generated from a random distribution and analyze standard key-comparison algorithms using this assumption [VCFF09].

Both approaches are unacceptable in certain cases: the first one - since often standard sorting algorithms are used for string sorting (such as the `sort` command in UNIX or in programming languages like C++, Java). The second approach is unsatisfying because often the input data does not come from a random distribution.

In [Sei10] it is argued that when sorting a set of  $n$  strings  $S$  using the lexicographical comparison, the entire prefix structure of the strings (or the prefix trie) should be used to parametrize the complexity of sorting. Hence we can apply the term “data-sensitive” sorting as the running time depends on the instance  $S$ .

The following definitions will help to arrive to the precise problem statement:

*Definition 1.* For a set of strings  $S$  the prefix trie  $T(S)$  is a trie that has all the prefixes of strings in  $S$  as its nodes. A string  $w$  is a child of another string  $v$  iff  $v$  is one symbol shorter than  $w$  and  $v$  is a prefix of  $w$ .

*Definition 2.* Denote  $\vartheta(w)$ , the thickness of the node  $w$  - the number of leaves under the node  $w$ .

Intuitively  $\vartheta(w)$  is the number of strings that have  $w$  as a prefix. Note that the prefix trie  $T(S)$  is not used as a data structure but merely as a parametrization for the running time of the algorithm.

*Definition 3.* Define the *reduced trie*  $\hat{T}(S)$  as the trie induced by the node set  $\{w \in P(S) \mid \vartheta(w) > 1\}$ .

A *reduced trie* simply discards strings which are suffixes of only themselves. In [Sei10] it is suggested that the vector  $(\vartheta(w))_{w \in P(S)}$  is a good parametrization choice and seems to determine the complexity of sorting.

Here are the results presented in [Sei10] that are relevant to the results that are to be shown in this thesis: Denote  $H_n = \sum_{1 \leq i \leq n} 1/i \approx 1 + \log(n)$ .

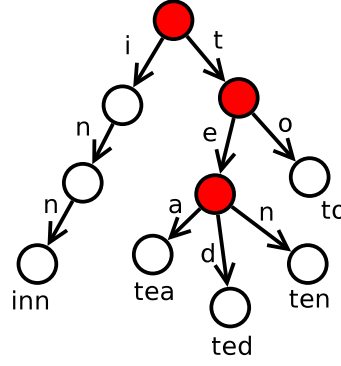


Figure 1.1: A reduced trie example

*Theorem 1.* Let  $Q(n) = 2(n+1)H_n - 4n$  be the expected number of key comparisons performed by Quicksort on a set of  $n$  keys. Assume Quicksort is applied to string set  $S$  employing the standard string comparison procedure. Then the expected number of symbol comparisons performed is exactly

$$\sum_{w \in P(S)} Q(\vartheta(w)) \quad (1.1)$$

The expression above is minimized when the trie  $T(S)$  is highly balanced. Based on this fact one can show that the expected number of comparisons is at least  $n \log^2(n) / \log(a)$ . A similar result holds for Mergesort by simply substituting  $Q$  by  $R$ , where  $R(n) = n \log_2(n)$ . Note that this already implies that it is *suboptimal* to use a generic algorithm to sort strings (as there are asymptotically better results [BS97]), however this does not discourage the analysis of such a parametrization, on the contrary, as most of the time exactly the generic algorithms are used. An advantage of such a parametrization is that it allows to compare sorting algorithms relative to each other on a given input instance.

The essence of the proof of the theorem 1 (see proof of theorem 1.1 in [Sei10]) above is captured by the following lemma: Call a comparison algorithm  $f(t)$ -*faithful*, when sorting a set  $X$  of randomly permuted keys  $x_1 < x_2 \dots < x_N$ , for any  $0 < t \leq N$  and for any “range”  $X_{(i,i+t]} = \{x_{i+1}, \dots, x_{i+t}\}$  of  $t$  order-consecutive elements of  $X$  the expected number of comparisons of the elements of  $X_{(i,i+t]}$  is at most  $f(t)$ . Call it *strongly*  $f(t)$  faithful if this expected number of comparisons among elements  $X_{(i,i+t]}$  is *exactly*  $f(t)$ .

*Lemma 1.* If an  $f(t)$ -faithful (strongly  $f(t)$  faithful) sorting algorithm is used to sort a set  $S$  of randomly permuted strings using the standard string comparison procedure, then the expected number of symbol comparisons is at most

$$\sum_{w \in P(S)} f(\vartheta(w)) \quad (1.2)$$

The lemma 1 allows us to abstract from analysis of sorting strings and instead focus on sorting general keys (in case of strings - symbols). The intuition behind the theorem, is that the algorithm does few comparisons overall if for a range of the elements that appear consecutive in the order few comparisons is done.

## 1.2 Problem statement

The motivation for this thesis is to give a similar analysis for Heapsort. Ideally we would like to get a result similar to the one in the theorem 1, namely, the expected number of symbol comparisons when using Heapsort on a set of  $n$  string  $S$  to be of the form:

$$\sum_{w \in P(S)} B(\vartheta(w)) \tag{1.3}$$

Where  $B(k)$  is the time required for Heapsort to sort  $k$  keys. Note that the result would only be interesting if the functions  $Q(k)$ ,  $B(k)$  and  $R(k)$  differ by a constant factor. As only then we can compare the algorithms relative to each other in an obvious way. Interest to Heapsort is well deserved since it is an in-place asymptotically worst-case optimal algorithm for sorting in a “key-comparison” based model, unlike most versions of Quicksort and Mergesort. Although the algorithm is asymptotically optimal, we will focus on getting the right constants in the analysis.

In order to show this, we need to demonstrate that for any subrange of the consecutive in-order elements few comparisons is done (ideally  $n \log_2(n)$ ).

## 1.3 Overview

This thesis is divided into two main parts: analysis for Heapsort using the standard binary heap data structure 2 and the one with a binomial heap [Vui78] as a heap implementation 3. In both section algorithms will be broken into stages. Both parts will consider the stages of building the heap and the actual sorting stage. In the beginning of each part we will show why randomization is needed in order to achieve useful bounds on the number of comparisons.

For *Binary Heapsort* we will firstly consider a different model of building the heap and prove bounds in this model, but later in the chapter the standard way of building the heap will be analyzed, and tight coupling between the two models will lead to the main result.

## Chapter 2

# Heapsort analysis

### 2.1 Overview

Let us firstly recall the original Heapsort algorithm [Wil64]. The algorithm sorts a set, stored in an array, of  $n$  keys using the binary heap data structure. Throughout this thesis, assume that all the heaps are *max*-heaps and that all logarithms are base 2, unless otherwise mentioned.

We assume that the heap is implemented as an array with indexes starting from 1. The navigation on the heap can be accomplished with the following procedures:

```
1: function LEFT( $a$ )
2:   return  $2a$ 
1: function RIGHT( $a$ )
2:   return  $2a+1$ 
1: function PARENT( $a$ )
2:   return  $a/2$ 
```

*Definition 4.* The *heap order* is maintained when the value at a node is larger or equal to the values at the left and right children.

Call a subheap the set of all the elements under a particular node. As a subroutine to the algorithm, the procedure *MaxHeapify* is called on a node that has both its left and right subheaps have heap order, but possibly not the itself node and its children. The procedure restores the heap order under a particular node:

```
1: function MAXHEAPIFY( $A, i$ )
2:    $l \leftarrow \text{Left}(i)$ 
3:    $r \leftarrow \text{Right}(i)$ 
```

```

4:   if  $l \leq A.size$  and  $A[l] > A[z]$  then
5:        $largest \leftarrow l$ 
6:   else
7:        $largest \leftarrow r$ 
8:   if  $r \leq A.size$  and  $A[r] > A[largest]$  then
9:        $largest \leftarrow r$ 
10:  if  $largest \neq i$  then
11:      swap  $A[i]$  with  $A[largest]$ 
12:       $MaxHeapify(A, largest)$ 

```

Having the *MaxHeapify* procedure at hand, the *BuildHeap* procedure is straightforward: *MaxHeapify* is called on all the nodes, starting from the leaves of the heap in a bottom-up fashion.

```

1: function BUILDHEAP( $A$ )
2:    $A.heapsize = A.size$ 
3:   for  $i = A.size$  downto 1 do
4:        $MaxHeapify(A, i)$ 

```

Once the heap is built, we can run the actual Heapsort algorithm by *Popping* the root of the heap and recursively *Sifting up* the larger of roots of the left and right subheaps using the *MaxHeapify* procedure.

```

1: function HEAPSORT( $A$ )
2:    $BuildHeap(A)$ 
3:    $A.heapsize = A.size$ 
4:   for  $i = A.size$  downto 2 do
5:       swap  $A[1]$  with  $A[i]$ 
6:        $A.heapsize = A.heapsize - 1$ 
7:        $MaxHeapify(A, i)$ 

```

In this thesis, we will give analysis for a version of the *Heapsort* algorithm with the Floyd's improvement [Flo64]. In [SS93] it is shown that this version is average-case optimal even up to constants, unlike the original version of the algorithm.

The modification of the algorithm lies in the *MaxHeapify* procedure. Call a node *unstable* if it is currently being *sifted down* (see element  $i$  in 1). In the original version of the algorithm, the children of the *unstable* node are compared and the larger one is compared to the *unstable*, hence yielding 2 comparisons each time the *unstable* element is moved one level down in the heap. Since (see line 5 in 1) the *unstable* element is taken from the



very bottom of the heap, it is likely that it will have to be *sifted* all the way *down* the heap.

In the Floyd's modification of the algorithm, the root node is not replaced with the right-most element, but instead after the maximal element is popped a *hole* is formed. During the *SiftDown* the larger of the two children of the *hole* is promoted up. Once the *hole* is at the bottom-most level of the heap, we put the *right-most* heap element to the *hole* position, and possibly *sift* it up to restore the *heap order*.

1: <b>function</b> SIFTDOWN( $A, i$ )	1: <b>function</b> SIFTUP( $A, i$ )
2: $l \leftarrow \text{Left}(i)$	2: <b>if</b> $i \neq 1$ <b>then</b>
3: $r \leftarrow \text{Right}(i)$	3: $p \leftarrow \text{Parent}(i)$
4: <b>if</b> $r \leq A.\text{heapsize}$ <b>and</b> $A[l] < A[r]$	4: <b>if</b> $A[p] < A[i]$ <b>then</b>
<b>then</b>	5: <b>swap</b> $A[p]$ <b>and</b> $A[i]$
5: <b>swap</b> $A[r]$ <b>and</b> $A[i]$	6: $\text{SiftUp}(A, p)$
6: <b>return</b> $\text{SiftDown}(A, r)$	
7: <b>else</b>	
8: <b>if</b> $l \leq A.\text{heapsize}$ <b>then</b>	
9: <b>swap</b> $A[l]$ <b>and</b> $A[i]$	
10: <b>return</b> $\text{SiftDown}(A, l)$	
11: <b>return</b> $i$	

The reason for efficiency of the modified version of the algorithm is that when the *SiftUp* is called, the element being *sifted up* was a leave of the heap and hence it is intuitively not likely to go high up the heap. On the other hand, in the original version of the algorithm we need to make 2 comparisons per level of recursion and we always need to descend to the leaves of the heap.

```

1: function MAXHEAPIFYFLOYD( $A, i$ )
2:    $j \leftarrow \text{SiftDown}(A, i)$ 
3:   swap  $A[A.\text{heapsize}]$  and  $A[j]$ 
4:    $\text{SiftUp}(A, j)$ 

```

For a more detailed analysis of the algorithm see [SS93].

The following part of the chapter is logically divided into two parts. In the first part, we consider the process of building a heap and count the number of comparisons this stage of the algorithm makes. In the second part the sorting part of the algorithm will be analyzed.

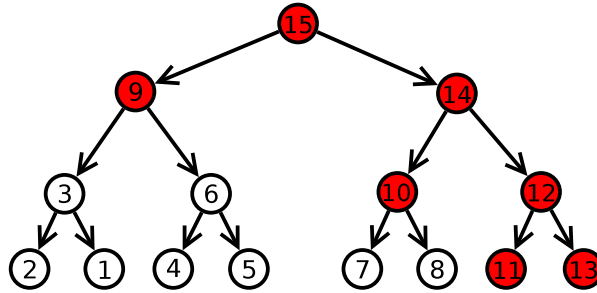
## 2.2 Randomization

In order to analyze performance of the *Heapsort* algorithm, let us define the setting more rigorously: let  $X = [x_1, x_2, \dots, x_n]$  such that  $x_i \leq x_{i+1}$  for all  $1 \leq i < n$ ,  $X_{i,i+r-1} = [x_i, \dots, x_{i+r-1}] \subseteq X$  be an order consecutive subrange of  $X$ .

In short, we would like to count the number of comparisons between the elements of  $X_{i,i+r-1}$  (possibly independent of  $i$ ) made by the *Heapsort* algorithm when executed on  $X$ . We will frequently use notation  $X_r$  (omitting the index  $i$ ) as most the results apply to any  $i$ .

When we talk about comparisons, only the comparisons between the elements of the set  $X_r$  are accounted for. For convenience, let us denote the  $X_r$  elements as **red** elements and the elements  $X \setminus X_r$  as **blue** elements.

Before diving into analysis, let us consider the following example: let the set  $X = [1, 2, \dots, n]$  and the set  $X_{2\lfloor \log(n) \rfloor + 1} = [n - 2\lfloor \log(n) \rfloor, \dots, n]$  for  $n = 2^k - 1$  for some  $k$ . In the example below  $k$  is 4.



**Figure 2.1:** A heap example. The set  $X_r$  is in red.

It is not hard to see that we can place the elements  $X_r$  as shown in the example above, namely so that the elements of  $X_r$  occupy the right spine of the heap and each one (except for the root) have a left neighbor coming from  $X_r$ .

*Claim 1.* The number of comparisons among elements of  $X_r$  done by Heapsort is  $O(r^2)$ .

*Proof.* The Floyd's modification algorithm would have to go to the end of the right spine of the heap with every pop operation, hence causing  $r/2$  operations on the first *sift down*,  $r/2$  on the second and  $\lfloor r/2 \rfloor - 1$  on the third etc.  $\square$

The proposed solution is to randomly shuffle the initial array of the elements  $X$ . It is well known that a uniform permutation of an array can be obtained in  $O(|X|)$  time (see [CLRS09] page 124). A "uniform" heap is highly unlikely to have such a degenerate distribution of the elements. A rigorous definition of a "uniform" heap is yet to come further in the chapter.

*Definition 5.* A uniformly at random permuted array  $A_\sigma$  is an array such that every permutation of  $A$  occurs equally likely in  $A_\sigma$ .

Since we are working with a uniformly random permutation of the initial array, the bounds on the number of comparisons are *expected*, although the *Heapsort* is a fully deterministic algorithm.

Another crucial aspect of the analysis is the so called “randomness preservation”. Intuitively, this property guarantees that given a “random” structure like heap, a removal of an element from the heap leaves the heap “random”. That is, after removal of an element, every heap configuration of a heap is equally likely. We will give a more detailed insight into this property. In the next chapter we demonstrate how the procedure *BuildHeap* affects the distribution of the elements in the heap and, in particular, show that the procedure *BuildHeap* is randomness preserving.

## 2.3 Building the heap

Let us show the effect of the *BuildHeap* on the randomness of distribution of elements  $X$ . Before we proceed we need to define what is a random heap rigorously:

*Lemma 2.* Let us call a heap uniformly random if every valid heap configuration occurs with same probability.

The claim is that if the original permutation is uniformly random, then the heap after running *BuildHeap* is also uniformly random, i.e. any heap configuration on  $X$  occurs with the same probability.

More specifically, we will now show that the probability of a particular heap on  $m$  elements occurring is  $1/H_m$ , where  $H_m$  is defined as:

$$H_0 = H_1 = 1 \tag{2.1}$$

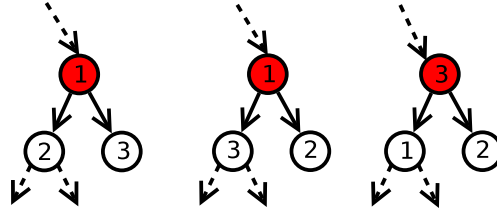
$$H_m = H_k H_{m-1-k} \binom{m-1}{k} \tag{2.2}$$

where  $k, m-k$  is the number of elements in the left and right heaps respectively. This would also imply that for a random heap, both its left and right subheaps are uniformly random as well.

*Proof.* Firstly, note that during heap construction, an element can only move on its path to the root of the heap or descend into the subheap it was initially placed by the outcome of permutation of  $X$ .

Suppose an element  $x$  is currently being *sifted down*. Let  $m$  be the number of elements strictly below  $x$  and  $k/(m - k)$  be the number of elements in the left/right subheaps respectively. Lets show inductively on the size of the heap that after a *SiftDown* we obtain a uniformly random heap. The base of the induction is clear.

During a *sift down* of  $x$ , 3 cases can occur: element  $x$  is swapped with one of its children (right or left) stays on its position.



**Figure 2.2:** 3 cases leading to a different *sift down* sequence:  $x = 1$ ,  $x = 1$ ,  $x = 3$

The probability that the maximal element in the heap of size  $m + 1$  is in the left/right subheaps is  $k/(m + 1)$  and  $(m - k)/(m + 1)$  respectively. The probability that the maximal element is  $x$  itself is  $1/(m + 1)$ . Let  $H_k$  be the number of possible heaps of size  $k$  on a fixed set of elements, then the probability of a particular heap occurring is  $1/H_k$ . Hence the probability of a particular (left,right) pair of subheaps is  $1/(H_k H_{m-k} \binom{m}{k})$ , as choosing  $k$  elements for the left subheap completely determines the elements of the right subheap.

By induction hypothesis, it holds that the the probability of a particular heap on  $m + 1$  elements (and by construction from procedure *BuildHeap*) is:

$$\frac{1}{m+1} \frac{1}{(H_k H_{m-k} \binom{m}{k})} + \frac{k}{m+1} \frac{1}{(H_k H_{m-k} \binom{m}{k})} + \frac{m-k}{m+1} \frac{1}{(H_k H_{m-k} \binom{m}{k})}$$

Simplifying the expression gives

$$H_{m+1} = H_{m-k} H_k \binom{m}{k} \quad (2.3)$$

which is exactly the induction hypothesis. Hence, the inductive step holds and the proof is complete. A different proof can be found in [Knu73, p. 153].  $\square$

Having the previous lemma in hand, we can proceed to analysis of the asymptotic behavior of the expression describing the number of comparisons made between elements  $X_r$  during the *BuildHeap* procedure.

One of the main results in this thesis is the following lemma:

*Lemma 3.* (Number of comparisons during construction of the heap)

The number of comparisons during construction of the heap between the elements of subrange  $X_r$  is  $O(r)$

*Proof.* For the sake of simplicity, let us assume that the number of elements in the heap is of the form  $m = 2^k - 1$ . The following argument extends to heaps of different size, at the expense of making the proofs rather technical.

*Definition 6.* (Expected number of comparisons caused by a single element) Let  $C(m, r)$  be the expected number of comparisons between the **red** elements caused during a *sift down* of the *root* of a heap having  $m$  elements and containing the entire  $X_r$ .

*Definition 7.* (Expected number of comparisons in a heap)  $T(r, m)$  is the expected number of comparisons between the **red** elements performed to build a subheap of size  $m$ , given that elements of  $X_r$  lie in the subheap

Having the two definitions at hand, we can relate them as follows:

Let

$$P_{2m+1, r, r'} = \frac{\binom{2m+1-r}{m-r'} \binom{r}{r'}}{\binom{2m+1}{m}} \quad (2.4)$$

be the probability of a particular split of elements of  $X_r$  to the left and the right subheaps, such that there are  $r'$  elements in the left heap and  $r - r'$  in the right one, then

$$T(2m+1, r) = \sum_{r \geq r' \geq 0} P_{2m+1, r, r'} (T(m, r') + T(m, r)) + C(2m+1, r) \quad (2.5)$$

One thing to notice is that if the *root* is in  $X_r$ , then the number of elements to distribute to left and right subheap is  $r - 1$  and not  $r$ , as stated in the recurrence relation. However, the following observation helps:

*Observation 1.* (The number of comparisons is an increasing function in  $r$ )

$$T(2m+1, r) \geq T(2m+1, r') \text{ for } r \geq r' \quad (2.6)$$

*Proof.* We can restrict counting the comparisons only for  $r'$  largest elements among the  $r$  **red** elements. □

And hence we can think that there are in fact  $r$  elements to distribute to the subheaps.

To establish  $C(m, r)$ 's asymptotic behavior it is useful to observe that

*Observation 2.* (Elements  $X_r$  form connected subheaps) On a path between two **red** elements, such that one is an ancestor of the other, there are no **blue** elements. This implies that the **red** elements form connected, possibly disjoint, subheaps after the *BuildHeap* procedure.

*Proof.* Consider an element  $x \in X_r$  in a heap. If there are is an element  $x' \in X_r$  “under”  $x$ , then there cannot be any elements  $\bar{x} \in X \setminus X_r$  on the internal path from  $x$  to  $x'$  as then this would contradict the heap order.  $\square$

The previous observation suggests that an estimate for  $C(m, r)$  is roughly the “average” depth of a subheap formed by elements of  $X_r$ . We can formulate this relation as follows:

*Lemma 4.* (Expected number of comparisons during a *sift down*) Let

$$P_{2m+1, r, r'} = \frac{\binom{2m+1-r}{m-r'} \binom{r}{r'}}{\binom{2m+1}{m}} \quad (2.7)$$

be the split probability in a heap of size  $m$ , containing entire  $X_r$ , such that the left subheap contains  $r'$  elements of  $X_r$  and the right subheap  $r - r'$ .

The expected number of comparisons created by an element *sifted down*  $C(m, r)$  can be upper bounded by

$$C(2m+1, r) \leq \sum_{r > r' \geq 0} P_{2m+1, r-1, r'} \left( \frac{r'}{r-1} C(r', m) + \frac{r-r'-1}{r-1} C(r-r'-1, m) \right) + 2 \quad (2.8)$$

and

$$C(2m+1, r) \leq 16 \log(r) \quad (2.9)$$

*Proof.* Let an element  $x \in X_r$  be *sifted down*. By the observation above, the elements of  $X_r$  form connected subheaps. On its way down,  $x$  will enter one of those connected subheaps, and possibly cause one extra comparison of two roots of the subheaps to decide which one to descend to. The number of such comparisons is at most  $O(r)$ . When  $x$  is *sifting down*, we need 2 comparisons to decide if it should be placed lower than the current position. Note that if  $x \notin X_r$ , then at most one comparison is needed.

We can now bound  $C(m, r)$  assuming element  $x$  descended to a subheap containing  $r$  elements, as any of the subheaps in which  $x$  descends has size at most  $r$ . Note that, when descending  $x$  will swap with the smaller of its two children. The probability that the smaller element is in the left or the right subheap is  $r'/(r-1)$  and  $(r-r'-1)/(r-1)$  respectively, hence the expression above.

Let us suppose that  $C(r', m') \leq 16 \log(r')$  for all  $m' \leq m, r' \leq r$  and inductively show that  $C(r, m) \leq 16 \log(r)$ . The base case is clear, as for  $r \leq 2$  we can have only 1 comparison of elements of  $X_r$ .

It is not hard to show that  $P_{m,r-1,r'}$  is increasing for  $r' \leq (r-1)/2$  and decreasing for  $r' \geq (r-1)/2$ , the opposite holds for  $r' \log(r) + (r-r'-1) \log(r-r'-1)$ . Applying the induction hypothesis to the first equation in the statement of the lemma and the rearrangement inequality:

$$\sum_{r>r'\geq 0} P_{2m+1,r-1,r'} \left( \frac{r'}{r-1} C(r', m) + \frac{r-r'-1}{r-1} C(r-r'-1, m) \right) \leq \quad (2.10)$$

$$\sum_{r>r'\geq 0} P_{2m+1,r-1,r'} \left( \frac{16r'}{r-1} \log(r') + \frac{16(r-r'-1)}{r-1} \log(r-r'-1) \right) \leq \quad (2.11)$$

$$\left( \sum_{r>r'\geq 0} P_{2m+1,r-1,r'} \right) \left( \sum_{r>r'\geq 0} \frac{16r'}{(r-1)^2} \log(r') + \frac{16(r-r'-1)}{(r-1)^2} \log(r-r'-1) \right) \quad (2.12)$$

In the last inequality, there are  $r$  terms, but we divide the expression by  $r-1$  instead of  $r$ , which only makes it larger. In the appendix we show that, for  $r \geq 16$ :

$$\sum_{r>r'\geq 0} r' \log(r') < \frac{(r-1)^2}{2} \log(r) - \frac{(r-1)^2}{16} \quad (2.13)$$

Notice that  $\sum_{r>r'\geq 0} P_{2m+1,r-1,r'} = 1$ . Combined with the bound above, we get that for  $r \geq 16$ :

$$(12) \leq 16 \log(r) - 2 \quad (2.14)$$

which is exactly what was needed to prove. Note that, for any  $m$  and  $r$ ,  $C(m, r) \leq r$  and so  $C(m, r) \leq 16 \log(r)$  for  $r < 16$  and the induction step is complete. It is possible to reduce the constant 16 further, as the expense of increasing the complexity of analysis. The point of the proof was to demonstrate that the number of comparisons can be bounded by  $C \log(r)$  for some constant  $C$ .  $\square$

Returning to the analysis of asymptotics of  $T(m, r)$ , what we wanted to show is that  $T(m, r) = O(r)$  for all  $m$ . We can show this inductively by proving that

$$T(r, 2m+1) \leq c_1 r - c_2 \log(r) - c_3 \quad (2.15)$$

for some constants  $c_1, c_2, c_3$ , for all  $m$  and  $r$ . Clearly, we can find such constants  $c_i$  to make the base of the inductive claim hold. Suppose for all  $m' < m, r' < r$ , the statement holds, then plugging the inductive hypothesis to the statement of the lemma gives:

$$T(2m+1, r) = \sum_{r \geq r' \geq 0} P_{2m+1, r, r'} (T(m, r') + T(m, r - r')) + C(2m+1, r) \leq (2.16)$$

$$\sum_{r \geq r' \geq 0} P_{2m+1, r, r'} (T(m, r') + T(m, r - r')) + 16 \log(r) \leq (2.17)$$

$$\sum_{r \geq r' \geq 0} P_{2m+1, r, r'} (c_1 r - c_2 (\log(r') + \log(r - r')) - 2c_3) + 16 \log(r) \quad (2.18)$$

By definition,  $\sum_{r \geq r' \geq 0} P_{2m+1, r, r'} = 1$ . Then what we need to show is that

$$\sum_{r \geq r' \geq 0} P_{2m+1, r, r'} (c_1 r - c_2 (\log(r') + \log(r - r')) - 2c_3) + 16 \log(r) \leq (2.19)$$

$$c_1 r - c_2 \log(r) - c_3 \quad (2.20)$$

Simplifying the expression gives

$$- P_{2m+1, r, r'} (c_2 (\log(r') + \log(r - r')) + 2c_3) + 16 \log(r) \leq (2.21)$$

$$-c_2 \log(r) - c_3 \quad (2.22)$$

Observe that both  $P_{2m+1, r, r'}$  and  $\log(r') + \log(r - r')$  are increasing for  $r' \in [1, r/2]$  and decreasing for  $r' \in (r/2, r]$  hence we can apply the rearrangement inequality

$$\begin{aligned} \sum_{0 \leq r' \leq r} P_{2m+1, r, r'} (\log(r') + \log(r - r')) &\geq \\ \frac{(\sum_{0 \leq r' \leq r} P_{2m+1, r, r'})}{r+1} \sum_{0 \leq r' \leq r} (\log(r') + \log(r - r')) &= \frac{2 \log(r!)}{r+1} \end{aligned}$$

It is well known that

$$e \left( \frac{n}{e} \right)^n \leq n! \Rightarrow n \log(n) - n \log(e) + \log(e) \leq \log(n!) \quad (2.23)$$

Plugging the lower bound to the inequality (22), leads to



$$16 \log(r) \leq \frac{2c_2}{r+1} (r \log(r) - r \log(e) + \log(e)) - c_2 \log(r) + c_3 \quad (2.24)$$

$$= (c_2 r \log(r) - 2c_2 r \log(e) + 2c_2 \log(e)) / (r+1) + c_3 \quad (2.25)$$

Taking  $c_2 = 16$  and  $c_3 = 3c_2$  makes the statement hold and hence the inductive step is complete. We have finally established that  $T(m, r) = O(r)$  for any  $m$ .  $\square$

## 2.4 Different model of building heap

Coming to the second part of analysis, we need to count the number of comparisons made by *Heapsort* after the heap is built. Let us define the concept of a “premature” heap.

*Definition 8.* (Premature heap) A heap is called *premature* if none of the **red** elements is at the top of the heap. We will call a heap *mature* when it is not *premature* anymore.

In case the largest **red** element is at the top of the heap, all the **red** elements are still in the heap and form a connected tree.

Let us count the expected number of comparisons once the heap is *mature* using a slightly different model of building heap. Namely, when a new element  $x$  is inserted into the current heap

- if  $x$  is larger than the current *root*, put  $root = x$  and reinsert *root*
- in case  $x$  is smaller or equal to *root*, insert  $x$  into the left or right subheaps of the root with  $p = 1/2$ .

We are about to show that the presented model of building the heap is randomness preserving, unlike the model of building the heap using the *BuildHeap* procedure.

This model allows one to analyze the number of comparisons “easily” and, as later will be shown, our previous model of heap construction via random permutation of the initial sequence will have smaller expected number of comparisons of elements in  $X_r$ .

Consider a heap on  $N$  elements built in this way. Notice that a consequence of this model is that a certain set of elements has probability  $p_k^N = \binom{N-1}{k} / 2^{N-1}$  of occurring in the left subheap of the root.

*Lemma 5.* (Preserving randomness)

After a *PopMax* operation any heap on  $N + 1$  elements has the same distribution as if it was built from scratch by taking all the elements out and reinserting them. More precisely, the probability that a particular set lies in the left subheap of the root is  $p_k^N = \binom{N-1}{k} / 2^{N-1}$ .

*Proof.* Let us call *split* a certain set of the elements in the left subheap (this set fully identifies the elements in the right subheap). Then  $p_k^N$  is the probability of a particular split on  $N$  elements, such that there are  $k$  elements in the left subheap. Such a split could have resulted after a *PopMax* operation on a heap with  $N + 1$  nodes in total and

1.  $k$  nodes in the left subheap and the next largest element being in the right subheap
2.  $k + 1$  nodes in the left subheap and the next largest element being in the left subheap

Then, in the first case the probability that the second largest element is in the right subheap is  $(N - k)/N$  similarly for the second case and the left subheap the probability is  $(k + 1)/N$ .

In order to demonstrate randomness preservation, what we need to show is that

$$p_k^N = \frac{k + 1}{N} p_{k+1}^{N+1} + \frac{N - k}{N} p_k^{N+1} \quad (2.26)$$

Rewriting the above equation, we have to show that  $\frac{\binom{N-1}{k}}{2^{N-1}} = \frac{k + 1}{N} \frac{\binom{N}{k+1}}{2^N} + \frac{(N - k)}{N} \frac{\binom{N}{k}}{2^N}$ , or, equivalently  $\frac{\binom{N-1}{k}}{2^{N-1}} = \frac{\binom{N-1}{k}}{2^N} + \frac{\binom{N-1}{k}}{2^N}$ , which clearly holds.  $\square$

*Lemma 6.* (Expected number of comparisons once the heap is *mature*) The expected number of comparisons **red** elements once the heap is *mature* can be upper bounded by  $r \log r$  under the model just described.

*Proof.* Taking into account one of our previous observations, the elements of  $X_r$  occupy a connected tree in the heap. As the root of the heap comes from  $X_r$  and elements  $X_r$  create a connected tree and hence the number of comparisons between the **red** elements is not affected by the elements from  $X/X_r$ . Intuitively, we can remove all the elements in  $X/X_r$  and both distribution of the elements in  $X_r$  and the number of comparisons between the **red** elements will not be affected. Based on these facts we can devise a recursive expression for the number of comparisons made between the elements of  $X_r$ :

Without loss of generality (based on the previous observation) we can take  $r = N$ . Let  $C_N$  be the number of comparisons for a subrange  $X_N$  of size  $N$ , created by an element *sifted down*. Then by induction we can derive the following:

1. Clearly  $C_0, C_1, C_2 = 0$ .
2. Once an element is *popped* from the heap, we execute the recursive *SiftDown* procedure. The two children of the root are compared only if both of the right or the left subheaps are non-empty. This happens with probability  $1 - p_{N-1}^N - p_0^N$ .
3. The probability that an element in the left subheap is larger than an element in the right subheap is  $k/(N-1-k)$  (similarly we can derive the probability for the right subheap).
4. Putting it all together we have:

$$C_N = 1 - p_{N-1}^N - p_0^N + \sum_{k=1}^{N-1} p_k^N \left( \frac{k}{N-1} C_k + \frac{N-1-k}{N-1} C_{N-1-k} \right)$$

Because of the symmetry the expression simplifies to the following:

$$C_N = 1 - p_{N-1}^N - p_0^N + \sum_{k=1}^{N-1} \binom{N-2}{k-1} C_k / 2^{N-2}$$

Instead of working with the recurrence directly, we will use an upper bound on the  $C_N$ , namely,

$$C_N \leq 1 + \sum_{k=0}^{N-2} \binom{N-2}{k} C_{k+1} / 2^{N-2}$$

We will analyze the behaviour of this recurrence using the exponential generating function  $g(z) = \sum_{k=0}^{\infty} C_k \frac{z^k}{k!}$ . Multiplying the recurrence with  $\frac{z^{N-2}}{(N-2)!}$  we have

$$\begin{aligned} \frac{C_N}{(N-2)!} z^{N-2} &= \frac{z^{N-2}}{(N-2)!} + \sum_{k=0}^{N-2} \binom{N-2}{k} \frac{C_{k+1}}{(N-2)!} \left(\frac{z}{2}\right)^{N-2} \\ &= \frac{z^{N-2}}{(N-2)!} + \sum_{k=0}^{N-2} \left(\frac{z}{2}\right)^{N-2} \frac{C_{k+1}}{k!(N-2-k)!} \\ &= \frac{z^{N-2}}{(N-2)!} + \sum_{k=0}^{N-2} \frac{C_{k+1}}{k!} \left(\frac{z}{2}\right)^k \frac{1}{(N-2-k)!} \left(\frac{z}{2}\right)^{N-2-k} \end{aligned}$$

Denoting  $m = N - 2$ , we have for  $m > 0$ :

$$\frac{C_{m+2}}{m!} z^m = \frac{z^m}{m!} + \sum_{k=0}^m \frac{C_{k+1}}{k!} \left(\frac{z}{2}\right)^k \frac{1}{(m-k)!} \left(\frac{z}{2}\right)^{m-k}$$

Now, notice that  $\sum_{k=1}^{m+1} \frac{C_k}{(k-1)!} \left(\frac{z}{2}\right)^{k-1} = g'(z/2)$  and  $\sum_{k=0}^m \frac{C_{k+1}}{k!} \left(\frac{z}{2}\right)^k \frac{1}{(m-k)!} \left(\frac{z}{2}\right)^{m-k}$  is the convolution of  $g'(\frac{z}{2})$  and  $e^{\frac{z}{2}}$ . Summing up all terms for all  $m$  on the left and right sides and taking into account that  $C_2 = 0$ , we have  $g''(z) = e^z - 1 + e^{z/2} g'(\frac{z}{2})$ .

Lets now show that  $g'(z) \leq g''(z)$  or equivalently  $\sum_{k=0}^{\infty} C_{k+1} \frac{z^k}{k!} \leq \sum_{k=0}^{\infty} C_{k+2} \frac{z^k}{k!}$ . This would clearly follow from the fact that  $C_k$  is increasing.

*Lemma 7.* (The sequence  $C_k$  is increasing)

*Proof.* Suppose that  $C_k$  is increasing for  $k < N$ . Let us show that  $C_N > C_{N-1}$ :

$$\begin{aligned} C_N &= 1 + \sum_{k=0}^{N-2} C_{k+1} \frac{\binom{N-2}{k}}{2^{N-2}} \\ C_{N-1} &= 1 + \sum_{k=0}^{N-3} C_{k+1} \frac{\binom{N-3}{k}}{2^{N-3}} \end{aligned}$$

Subtracting the two expressions we get

$$\begin{aligned} C_N - C_{N-1} &= C_{N-2} + \sum_{k=0}^{N-3} \left( \frac{\binom{N-2}{k}}{2^{N-2}} - 2 \frac{\binom{N-3}{k}}{2^{N-2}} \right) C_{k+1} \\ &= C_{N-2} + \sum_{k=0}^{N-3} \left( \frac{\binom{N-3}{k-1}}{2^{N-2}} - \frac{\binom{N-3}{k}}{2^{N-2}} \right) C_{k+1} \end{aligned}$$

The sum  $\sum_{k=0}^{N-3} ((\binom{N-3}{k-1}) - (\binom{N-3}{k})) C_{k+1}$  is non-negative, as  $\binom{N-3}{k-1} - \binom{N-3}{k}$  is symmetric around  $\frac{N-3}{2}$  and is positive for  $k \geq \lfloor \frac{N-2}{2} \rfloor$  and  $C_k$  is increasing by induction assumption for  $k \leq n-2$ . Hence  $C_N - C_{N-1} > 0$  and induction is complete.  $\square$

Coming back to the generating function identity  $g''(z) = e^z - 1 + e^{z/2} g'(z/2)$ , we showed that we can lower bound  $g''(z)$  with  $g'(z)$ , from which it follows that the following inequality holds:

$$g'(z) \leq g''(z) = e^z - 1 + e^{z/2} g'(z/2)$$

Let  $h(z) = g'(z)$ , iterating the above inequality we have:

$$h(z) \leq e^z - 1 + e^{z/2} (e^{z/2} - 1) + e^{z/2+z/4} (e^{z/4} - 1) \dots = \sum_{j=0}^{\infty} e^z - e^{z(1-1/2^j)}$$

Thus, the exponential generating function  $h(z)$  has coefficients  $H_k = \sum_{j=0}^{\infty} (1 - (1 - 1/2^j)^k)$ .

Expanding the expression using the fact that  $(1 - \frac{1}{2^j})^k = \exp(-k/2^j)(1 + O(1/k))$  gives:

$$H_k = \sum_{0 \leq j < \lfloor \log k \rfloor} 1 - e^{-k/2^j} + \sum_{j \geq \lfloor \log k \rfloor} 1 - e^{-k/2^j} + o(1) \quad (2.27)$$

$$= \lfloor \log k \rfloor - \sum_{0 \leq j < \lfloor \log k \rfloor} e^{-k/2^j} + \sum_{j \geq \lfloor \log k \rfloor} 1 - e^{-k/2^j} \quad (2.28)$$

$$= \lfloor \log k \rfloor - \sum_{j < \lfloor \log k \rfloor} e^{-k/2^j} + \sum_{j \geq \lfloor \log k \rfloor} 1 - e^{-k/2^j} + O(e^{-k}) \quad (2.29)$$

As  $\sum_{j < 0} e^{-k/2^j} \leq \sum_{j > 0} e^{-k/2^j} = e^{-k}/(1 - e^{-k}) - 1 < e^{-k}$ . Shifting the summations by  $\lfloor \log k \rfloor$ ,

$$\lfloor \log k \rfloor - \sum_{j < 0} e^{-k/2^{j+\lfloor \log k \rfloor}} + \sum_{j \geq 0} 1 - e^{-k/2^{j+\lfloor \log k \rfloor}} + O(e^{-k})$$

It is easy to establish that  $-\sum_{j < 0} e^{-k/2^{j+\lfloor \log k \rfloor}} + \sum_{j \geq 0} 1 - e^{-k/2^{j+\lfloor \log k \rfloor}} = O(1)$ . A more precise analysis of this function is available in [SS93].

□

What we have shown is that a *sift down* takes  $\log(r) + O(1)$  time, assuming the split distribution described above, and after each *PopMax* operation the heap randomness is preserved. Hence the total number of comparisons is  $r \log r + O(r)$  under the model described.

It is seemingly not easy to derive bounds for the number of comparisons of elements in  $X_r$  after the heap is build but before the heap is *mature*. Perhaps one could count the number of comparisons after the heap is built but before the heap is *mature*, which would complete our analysis. The analysis is however not futile, as we will use details of the section further in the thesis to show our main result.

## 2.5 Original way of building heap

Let us count the number of comparisons after the heap is built in a different way. For now assume that  $\forall x \in X_r$  is larger than the median of the entire range  $X$ . This assumption is implementable in the following way: after the heap is built, fill the lower most heap level (or possibly even the next level) with an element  $x'$  which is strictly smaller than all

the elements in  $X$ . What we need to achieve is that the number of additional elements on the lowermost level of the heap is larger than the size of  $X$ .

The assumption guarantees that when we possibly place the right most minimal element of the heap to the empty hole created by the *SiftDown* and execute *SiftUp*, we do not have additional comparisons of elements in  $X_r$ , as the element *sifted up* is smaller than elements in  $X_r$ .

One might notice that if we introduce  $n$  dummy elements to the heap, this would incur additional comparisons and we would be cheating in a way, as we only count the comparisons of the **red** elements and ignore comparisons of the dummy elements. However we will show later in this section, that dummy elements introduce only  $O(n)$  more comparisons overall to the running time of the algorithm, which is “acceptable”.

*Observation 3.* Consider a node  $x$  in the heap. Suppose during a *SiftDown* a comparison of elements  $left(x), right(x) \in X_r$  happens, in which case one is promoted. Then the total number of the elements  $\in X_r$  in the subheaps rooted at  $left(x), right(x)$  decreases by one.

Also, note that such a comparison can happen iff initially there were some elements  $\in X_r$  in subheaps rooted at  $left(x), right(x)$ . Hence, an element  $x$  should be “splitting” a subset of  $X_r$  into two subsets of size greater than zero.

*Observation 4.* There can be at most  $r - 1$  such positions  $x$ . Such positions can be identified by taking the union of all the lowest common ancestors of the pairs of elements of  $X_r$  in the heap.

Using the observation above we can express the maximum number of comparisons in terms of “splits” of the set  $X_r$  across the heap as follows:

*Lemma 8.* Consider a node  $x$  of the heap. Let it have  $p$  and  $q$  elements of  $X_r$  in the left and the right subheaps respectively. Then there can be at most  $p + q - 1$  comparisons of elements in  $X_r$  at positions  $left(x), right(x)$

*Proof.* Using the observation above, after each comparison, the number of elements of  $X_r$  in the subheaps of  $x$  decreases and we need at least one element of  $X_r$  in both right and left subheaps.  $\square$

*Observation 5.* The elements  $x \in X_r$  stay on their original paths to the root of the heap during the *SiftUp* procedure.

*Proof.* Now the assumption that the elements  $x \in X_r$  are larger than the median of the array  $X$  becomes handy. Note that during *SiftUp* procedure, the elements only move

up the path to the root of the heap and only the right-most element is possibly moved from its path to the root. With the assumption, elements of  $X_r$  do not appear in the lower-most level of the heap. Hence, the elements  $x \in X_r$  stay on their original paths to the root of the heap. From now on we will assume that the elements  $X_r$  do not lie in the lower most layer of the heap during running of the *Heapsort* algorithm, unless explicitly stated otherwise.  $\square$

*Definition 9.* (Number of comparisons  $C(m, r)$ ) Let  $C(m, r)$  be the number of comparisons between the elements  $x \in X_r$  in a heap of size  $m$  during the sorting phase of the algorithm.

Now, let us classify the comparisons of elements in  $X_r$  to make the computations easier.

*Definition 10.* (Good comparisons) Call a comparison “good” if it compares the roots of two subheaps of elements  $\in X_r$  and call all the other comparisons “bad”.

*Lemma 9.* (Bounding “good” comparisons)

There are at most  $r - 1$  “good comparisons”, where  $r = |X_r|$

*Proof.* Each time a “good” comparison happens, two subheap are merged. Once the subheaps of elements in  $X_r$  are merged, they stay connected. We can merge at most  $r$  separate subheaps and hence make at most  $r - 1$  “good” comparisons.  $\square$

From now on, we will only count “bad” comparisons. As “good” comparisons do not contribute to the asymptotic behavior of the number of comparisons more than a linear term.

*Observation 6.* Let there be  $p/q$  elements from  $X_r$  in the left/right subheaps of an element  $x$  respectively. Then there can be at most  $p + q - 1$  comparison at the node  $x$ , that is comparing elements at positions  $left(x)$ ,  $right(x)$ .

*Lemma 10.* (Upper bounding “bad” comparisons) The number of comparisons  $C(m, r)$  can be upper bounded by  $r \log r + O(r)$

*Proof.* Lets consider all the possible splits of the set  $X_r$  by the current *root* of the heap. As was shown above, there can be at most  $r - 1$  “bad” comparisons of elements  $X_r$  at a node, given that the node “splits” the set  $X_r$ , otherwise there will be no comparisons. Using the distribution of the elements of  $X_r$  in the heap:

$$C(2m + 1, r) \leq r - 1 + \sum_{r'=0}^r \frac{\binom{2m+1-r}{m-r'} \binom{r}{r'}}{\binom{2m+1}{m}} (C(m, r') + C(m, r - r'))$$

The recursive relation reminds the formula for counting the expected number of comparisons under a different model of building the heap discussed above. Now, let us exploit

the relation which we analyzed for a different model of building the heap. We have shown that for

$$C(0), C(1), C(2) = 0 \text{ and } C(N) = 1 + \sum \binom{N-2}{k} C(k+1) / 2^{N-2}$$

it holds that  $C_N = \log(N) + O(1)$ . Using a similar proof one can show that for a relation

$$G(0), G(1), G(2) = 0 \text{ and } G(N) = N + \sum \binom{N}{k} (G(k) + G(N-k)) / 2^N$$

it holds that  $G(N) = N \log(N) - O(N)$ . In [SS93] one can find a more precise analysis of the relation  $G(N)$ , in particular it is shown that  $G(N) < N \log(N) - \epsilon(N)N$ , where  $|\epsilon(N)| < 1^{-3}$ .

Coming back to the original expression, we wanted to show that for

$$C(2m+1, r) = r - 1 + \sum_{r'=1}^{r-1} \frac{\binom{2m+1-r}{m-r'} \binom{r}{r'}}{\binom{2m+1}{m}} (C(m, r') + C(m, r-r')) \quad (2.30)$$

it holds that  $C(m, r) \leq r \log(r)$ . What we will try to show is that  $C(m, r) \leq G(r)$  which is an even a stronger statement.

Intuitively, the probability distribution  $P_{m,r,r'} = \frac{\binom{2m+1-r}{m-r'} \binom{r}{r'}}{\binom{2m+1}{m}}$ , “encourages” even splits of the elements in  $X_r$  more than the split probability  $T_{m,r,r'} = \binom{r}{r'} / 2^r$  (of  $(G(r))$ ).

*Lemma 11.* (Split probability distributions) There exists a range  $[r/2 - \delta \dots r/2 + \delta]$  such that  $P_{m,r,r'} \geq T_{m,r,r'}$  for  $r' \in [r/2 - \delta \dots r/2 + \delta]$  and  $P_{m,r,r'} < T_{m,r,r'}$  otherwise.

*Proof.* To show this more rigorously, note that  $\sum_{r'} P_{m,r,r'} = 1$  and  $\sum_{r'} T_{m,r,r'} = 1$ , also, both  $T_{m,r,r'}$  and  $P_{m,r,r'}$  are increasing on  $r' \in [1, \frac{r}{2}]$ , however  $R_{m,r,r'}$  is a pointwise product of two binomial distributions. Note that if  $P_{m,r,r'} > T_{m,r,r'}$  then  $\frac{\binom{2m+1-r}{m-r'}}{\binom{2m+1}{m}} > \frac{1}{2^r}$ . And from the fact that  $P_{m,r,r'}$  is increasing on  $[0 \dots r/2]$  and both  $T_{m,r,r'}$ ,  $P_{m,r,r'}$  sum to 1 it follows that there exists such a  $\delta$ .  $\square$

*Corollary 1.*  $P_{m,r,r'} \geq T_{m,r,r'}$  for  $r' = 0$

The lemma above about the split probabilities combined with the next lemma will give us a way to analyze the relation  $C(m, r)$ . Intuitively the next lemma will show that the sequence  $G(k)$  is “concave”, i.e. the better the split of the elements  $X_r$ , the less comparisons happen. As the split probability  $P_{m,r,r'}$  “encourages” even splits more than the probability  $T_{m,r,r'}$ , we can bound  $C(m, r)$  by  $G(r)$ .



*Lemma 12.* (“Concavity” of the sequence  $G(N)$ )

For the sequence  $G(N)$  defined above, it holds that  $G(N - k) + G(k) \geq G(N - k') + G(k')$ , for  $|N/2 - k| > |N/2 - k'|$

*Proof.* Let us first show that  $G(N) - G(N - 1) \geq G(N - 1) - G(N - 2)$ . Using the solution to the recursion  $G(N)$  from [SS93], namely

$$G(N) = N \sum_{j \geq 0} (1 - \frac{1}{2^j})^{N-1} \quad (2.31)$$

we can simplify  $G(N + 2) - 2G(N + 1) + G(N)$  to

$$\dots = \sum_{j \geq 0} (-\frac{N+2}{2^{2j}} + \frac{2}{2^j}) (1 - \frac{1}{2^j})^{N-1} \quad (2.32)$$

Now we need to prove that the sum non-negative for every  $N$ . To show the non-negativeness, we will use the approximation of the sum with a definite integral (see appendix).

Let  $f(x) = \frac{1}{2^x} (1 - \frac{1}{2^x})^{N-1}$ , then

$$\int f(x) dx = \frac{(1 - \frac{1}{2^x})^N}{N \log_e(2)} \quad (2.33)$$

Similarly, let  $g(x) = \frac{1}{2^{2x}} (1 - \frac{1}{2^x})^{N-1}$ , then using product rule, we have

$$\int g(x) dx = 2^{-x} \frac{(1 - \frac{1}{2^x})^N}{N \log_e(2)} + \frac{(1 - \frac{1}{2^x})^{N+1}}{N(N+1) \log_e(2)} = \frac{(1 - \frac{1}{2^x})^N}{N \log_e(2)} - \frac{(1 - \frac{1}{2^x})^{N+1}}{(N+1) \log_e(2)} \quad (2.34)$$

Easy enough,  $\sum_{i \geq 0} 2f(i) - (N+2)g(i)$  is exactly the original sum. Now, let us split the original sum into negative and positive terms. Looking at multiple  $-\frac{N+2}{2^{2j}} + \frac{2}{2^j}$  appearing in the original summation, we conclude that terms for  $i \geq \log(N+2) - 1$  are non negative. We have to be slightly careful here, as for  $N+2 = 2^k$  for some integer  $k$  the term for  $i = \log(N+2) - 1$  is zero. Hence, to show the non-negativity of the original sum, we can simply show that

$$\sum_{j \geq \log(N+2)-1} (-\frac{N+2}{2^{2j}} + \frac{2}{2^j}) (1 - \frac{1}{2^j})^{N-1} \geq - \sum_{0 \leq j < \log(N+2)-1} (-\frac{N+2}{2^{2j}} + \frac{2}{2^j}) (1 - \frac{1}{2^j})^{N-1} \quad (2.35)$$

We are almost ready to apply the approximation of the sum by an integral. It is easy to establish that  $2f(x) - (N+2)g(x)$  is decreasing on  $j \in [0 \dots \log(N+2) - 1)$  and

similarly that  $2f(x) - (N+2)g(x)$  is decreasing on  $j \in [\log(N+2) - 1 \dots + \infty)$ . Hence, we can lower bound r.h.s and upper bound l.h.s as shown in the appendix by an integral and we would then need to show that

$$\sum_{j \geq \log(N+2)-1} \left(-\frac{N+2}{2^{2j}} + \frac{2}{2^j}\right) \left(1 - \frac{1}{2^j}\right)^{N-1}$$

$$\geq \int_{\log(N+2)-1}^{\infty} 2f(x) - (N+2)g(x) dx \quad (2.36)$$

$$\dots \geq \int_0^{\log(N+2)-1} 2f(x) - (N+2)g(x) dx \quad (2.37)$$

$$\dots \geq - \sum_{1 \leq j < \log(N+2)-1} \left(-\frac{N+2}{2^{2j}} + \frac{2}{2^j}\right) \left(1 - \frac{1}{2^j}\right)^{N-1} \quad (2.38)$$

$$\dots = - \sum_{0 \leq j < \log(N+2)-1} \left(-\frac{N+2}{2^{2j}} + \frac{2}{2^j}\right) \left(1 - \frac{1}{2^j}\right)^{N-1} \quad (2.39)$$

We can rewrite the inequality (17) as:

$$\int_{\log(N+2)-1}^{\infty} 2f(x) - (N+2)g(x) dx \geq \int_0^{\log(N+2)-1} (N+2)g(x) - 2f(x) dx \quad (2.40)$$

or

$$2 \frac{(1 - \frac{1}{2^x})^N}{N \log_e 2} - (N+2) \left( \frac{(1 - \frac{1}{2^x})^N}{N \log_e(2)} - \frac{(1 - \frac{1}{2^x})^{N+1}}{(N+1) \log_e(2)} \right) \Big|_{\log(N+2)-1}^{\infty} \geq \quad (2.41)$$

$$-2 \frac{(1 - \frac{1}{2^x})^N}{N \log_e 2} + (N+2) \left( \frac{(1 - \frac{1}{2^x})^N}{N \log_e(2)} - \frac{(1 - \frac{1}{2^x})^{N+1}}{(N+1) \log_e(2)} \right) \Big|_0^{\log(N+2)-1} \quad (2.42)$$

Now, note that  $g(x) \Big|_{x=0} = f(x) \Big|_{x=0} = 0$  and  $f(x) \Big|_{x \rightarrow \infty} = \frac{1}{N \log_e 2}$ ,  $g(x) \Big|_{x \rightarrow \infty} = \frac{1}{N \log_e 2} - \frac{1}{(N+1) \log_e 2}$ . We can get rid of the  $\log_e 2$  on both sides and rewrite the inequality as

$$2\frac{1}{N} - (N+2)\left(\frac{1}{N} - \frac{1}{N+1}\right) = \frac{1}{N} - \frac{1}{N(N+1)} = \frac{1}{N+1} \geq \quad (2.43)$$

$$2\left(-2\frac{(1-\frac{1}{2^x})^N}{N} + (N+2)\left(\frac{(1-\frac{1}{2^x})^N}{N} - \frac{(1-\frac{1}{2^x})^{N+1}}{(N+1)}\right)\right)\Big|_{x=\log(N+2)-1} = \quad (2.44)$$

$$2\left((1-\frac{1}{2^x})^N - (N+2)\frac{(1-\frac{1}{2^x})^{N+1}}{(N+1)}\right)\Big|_{x=\log(N+2)-1} = \quad (2.45)$$

$$2\left((1-\frac{1}{2^x})^N \frac{1}{2^x} - \frac{(1-\frac{1}{2^x})^{N+1}}{N+1}\right)\Big|_{x=\log(N+2)-1} \quad (2.46)$$

We can relax the inequality even further to

$$\frac{1}{N+1} \geq 2\left((1-\frac{1}{2^x})^N \frac{1}{2^x}\right)\Big|_{x=\log(N+2)-1} \quad (2.47)$$

To explore the behavior of a function  $h(x) = (1 - \frac{1}{2^x})^N \frac{1}{2^x}$  notice that

$$h'(x) = (1 - \frac{1}{2^x})^N (-\log_e 2) \frac{1}{2^x} + N(1 - \frac{1}{2^x})^{N-1} \frac{\log_e 2}{2^{2x}} \leq 0 \quad (2.48)$$

is equivalent to

$$N \leq (1 - \frac{1}{2^x})2^x = 2^x - 1 \leftrightarrow \log(N+1) \leq x \quad (2.49)$$

And, hence the function is increasing for  $\log(N+1) \leq x$  and attains its maximum at  $x = \log(N+1)$ . Hence,

$$2\left((1-\frac{1}{2^x})^N \frac{1}{2^x}\right)\Big|_{x=\log(N+2)-1} < 2\left((1-\frac{1}{2^x})^N \frac{1}{2^x}\right)\Big|_{x=\log(N+1)} \quad (2.50)$$

It is well known that  $(1 - \frac{1}{y})^y < \frac{1}{e}$  for  $y \geq 1$ . Hence, finally, we can relax the original inequality to

$$\frac{1}{N+1} > \frac{2}{e} \frac{1}{2^x}\Big|_{x=\log(N+1)} = \frac{2}{e} \frac{1}{N+1} \quad (2.51)$$

which clearly holds. Now, in order to show that  $G(N-k) + G(k) \geq G(N-k') + G(k')$ , for  $|N/2 - k| > |N/2 - k'|$  let without loss of generality  $N - k' > N - k > k > k'$ , then what we need to show is

$$G(N - k') - G(N - k) \leq G(k) - G(k') \quad (2.52)$$

but using telescoping sums we can rewrite it as

$$G(N - k') - G(N - k) = \quad (2.53)$$

$$\sum_{k' < i \leq k} (G(N - i) - G(N - i - 1)) \geq \sum_{k' < i \leq k} (G(i) - G(i - 1)) \quad (2.54)$$

$$= G(k) - G(k') \quad (2.55)$$

which holds as the sequence  $G(k)$  is convex.

As a sanity check, a simulating program for verifying “concavity” of  $G(N)$  was written. The program verified the fact for  $N \leq 10000$ . The proof actually shows us that  $G(N + 2) - 2G(N + 1) + G(N) > \frac{1 - \frac{2}{e}}{N + 1}$ , which is suggested by the results of the program as well.  $\square$

Using the above, let us show that  $C(m, r) \leq G(r)$  for all  $m = 2^k - 1$  inductively on  $r$ . The base of induction clearly holds (for  $r = 0, 1, 2$  and for all  $m$ ). Clearly,  $C(m, r) \leq G(r)$  for  $m < r$ . Using the fact that both  $G(r)$  and  $C(m, r)$  are increasing in  $r$  and the observation about the split probabilities we have

$$C(2m + 1, r) = \quad (2.56)$$

$$= r - 1 + \sum_{r'=0}^r \frac{\binom{2m+1-r}{m-r'} \binom{r}{r'}}{\binom{2m+1}{m}} [C(m, r') + C(m, r - r')] \quad (2.57)$$

$$\leq r - 1 + \frac{2 \binom{2m+1-r}{m}}{\binom{2m+1}{m}} C(m, r) + \sum_{r'=1}^{r-1} \frac{\binom{2m+1-r}{m-r'} \binom{r}{r'}}{\binom{2m+1}{m}} [G(r') + G(r - r')] \quad (2.58)$$

$$\leq r - 1 + \frac{2 \binom{2m+1-r}{m}}{\binom{2m+1}{m}} C(m, r) + \sum_{r'=1}^{r-1} \frac{\binom{r}{r'}}{2^r} [G(r') + G(r - r')] \quad (2.59)$$

$$\leq r - 1 + \frac{2}{2^r} G(r) + \sum_{r'=1}^{r-1} \frac{\binom{r}{r'}}{2^r} [G(r') + G(r - r')] \quad (2.60)$$

$$= r - 1 + \sum_{r'=0}^r \frac{\binom{r}{r'}}{2^r} [G(r') + G(r - r')] = G(r) \quad (2.61)$$

the first inequality comes from the induction hypothesis  $G(r') \geq C(m, r')$  for all  $m$  and  $r' < r$ . The second follows from the fact that the sequence  $G(r') + G(r - r')$  is decreasing (from the proof above) for  $r' \in [0, r/2]$  and the observation about the split probabilities. The third inequality follows from the observation about the split probabilities and induction hypothesis that  $C(m, r) \leq G(r)$ .

Hence,  $C(m, r)$  can be upper bounded by  $r \log(r)$  as well for  $m = 2^k - 1$ .  $\square$

Notice that the lemma above works only for the case where the initial number of elements is of the form  $2^k - 1$ . We can still make the *Heapsort* algorithm work as follows:

*Definition 11.* Call an expansion of a number  $n$  “almost binary” if

$$n = \sum_k (2^{s_k} - 1) \quad (2.62)$$

and

$$2^{s_k} - 1 \geq \sum_{i < k} 2^{s_i} - 1 \quad (2.63)$$

it follows that the sequence  $s_1, s_2, \dots, s_m$  is as small as possible and  $s_1 \geq s_2 \geq \dots \geq s_m$

It is easy to verify that such an expansion is unique for each  $n$ . But it could be that  $s_k = s_{k+1}$  for some  $k$ , for example when  $n = 2(2^k - 1)$ . Notice that, however, there is only one  $k$  for which such a condition holds, and  $s_k$  is the next to last in the sequence  $\{s_k\}_k$ . Because otherwise, let us take the smallest such  $i$ , but then, if such an  $k$  is not the last,  $2^{s_k} - 1 < 2^{s_{k+1}} - 1 + 1$  and hence the second condition of the definition of the “almost-binary” expansion doesn’t hold.

*Lemma 13.* The number of terms in such an expansion is at most  $\sum_{i=1} \log^{(i)}(n)$  where  $\log^{(i)}(n) = \log(n)$  for  $i = 1$  and  $\log^{(i)}(n) = \log(\log^{(i-1)}(n))$  for  $i > 1$ .

*Proof.* Denote the length of the almost-binary sequence of  $n$  as  $A(n)$ . Consider binary representation  $b_{ii}$  of  $n$ , which is known to have at most  $\log(n)$  terms. Then,  $n - \sum_i 2^{b_i} \leq \log(n)$  and hence,  $A(n) \leq A(\log(n)) + \log(n)$ . The statement of the lemma follows immediately.  $\square$

Having split the input elements to the subranges induced by the sizes of the “almost binary” expansion of  $n$ . As we have shown above, during *Heapsort* in each heap of size  $2^{s_k} - 1$  only  $r \log(r) - O(r)$  comparisons are created between the subrange elements. Suppose, that in a heap of size  $2^{s_k} - 1$ , there are  $r_k$  **red** elements, then the total number of comparisons in each heap is

$$\sum_k r_k \log(r_k) - O(r_k) \leq r \log(r) - O(r) \quad (2.64)$$

Once all of the heaps are sorted, we can merge them with a procedure analogous to the one in the classic *Mergesort*. We are only left to show that the number of comparisons of the **red** elements during this procedure is  $O(r)$ .

The following observations will help to establish the result:

*Observation 7.* If the initial permutation is permuted uniformly at random, then each subrange of size  $2^{s_k} - 1$  is also permuted uniformly, i.e. every subset of size  $2^{s_k} - 1$  of input elements is equally likely to appear in the subrange.

*Observation 8.* After a subrange of size  $2^{s_k} - 1$  is sorted, the **red** elements lie contiguously. This observation is similar to the one above, which says that elements of the range form a contiguous tree in a heap.

Let us now count the number of comparisons during the merge:

*Lemma 14.* To merge two arrays of size  $r_k$  and  $r_{k+1}$  we need at most  $r_k + r_{k+1} - 1$  comparisons of the **red** elements.

*Lemma 15.* During the merge procedure, there will be at most  $\sum_{1 \leq k \leq m} kr_k$  comparisons of the **red** elements.

*Proof.*  $k$ -th subrange's elements will be merged with the elements of subranges  $1, 2, \dots, k-1$  only.  $\square$

Denote  $r_k$  the expected number of **red** elements in  $i$ -th heap. In expectation, by the lemma above, we will have  $E(\sum_{1 \leq k \leq m} r_k k)$  comparisons. Exploiting linearity of expectation, we have

$$E\left(\sum_{1 \leq k \leq m} kr_k\right) = \sum_{1 \leq k \leq m} kE(r_k) = \sum_{1 \leq k \leq m} kr \frac{2^{s_k} - 1}{n} \quad (2.65)$$

We can rewrite the sum as a number of partial sums:

$$\sum_{1 \leq k \leq m} kr \frac{2^{s_k} - 1}{n} = \sum_{1 \leq k \leq m} \sum_{k \leq j \leq m} r \frac{2^{s_k} - 1}{n} \quad (2.66)$$

And by the property of the sequence  $s_k$  that

$$2^{s_k} - 1 \geq \sum_{i < k} 2^{s_i} - 1 \quad (2.67)$$

we can conclude that

$$\sum_{1 \leq k \leq m} kr \frac{2^{s_k} - 1}{n} = \sum_{1 \leq k \leq m} \sum_{k \leq j \leq m} r \frac{2^{s_k} - 1}{n} \leq \sum_{1 \leq k \leq m} r(n/2^{k-1})/n \leq 2r \quad (2.68)$$

And hence, during the merge procedure, the expected number of comparisons of the **red** elements is  $O(r)$  and the total number of comparisons during *Heapsort* between the **red** elements is  $r \log(r) + O(r)$ .

As was promised, earlier, we give a proof that the dummy elements only contribute  $O(n)$  comparisons to the overall number of comparisons of the algorithm:

There are 3 possible comparison types:

1. non-dummy and non-dummy
2. dummy and non-dummy
3. dummy and dummy

*Observation 9.* During the *Delete Max* operation, a dummy element moves up on its path to the root iff there was a comparison of two dummy elements.

*Observation 10.* If there was a comparison of a dummy and a non-dummy element, only the non-dummy element moves up the path to the root of the heap, as dummies are strictly smaller than non-dummy elements.

Hence, we can bound the total number of comparisons dummy elements cause with

1. cumulative path length of non-dummy elements to the root of the heap (comparisons of types 1, 2)
2. cumulative path length of dummy elements to their positions once *Heapsort* is done (comparisons of type 3)

We have already shown that for a range of size  $r$ , there are  $r \log(r) + O(r)$  comparisons between the red elements during the sorting procedure. Taking  $r = n$  we immediately get that comparisons of type 1 account for no more than  $n \log(n) + O(n)$  of all the comparisons. But information theoretic lower bound tells us that there are at least  $n \log(n) + O(n)$  comparisons in any algorithm in comparison-based model. Hence, there are only  $O(n)$  comparisons of type 2.

The following lemma will show that there are only  $O(n)$  comparisons of type 3:

*Lemma 16.* The cumulative path length of all the dummy elements to their positions in the heap is  $O(n)$

*Proof.* Once *Heapsort* is done, the number of dummy elements at depth  $\log(n) + 1$  is 1, at depth  $\log(n) - 2$  and so on:

$$\sum_{0 \leq k \leq \lceil \log(n) \rceil} (\lceil \log(n) \rceil - k + 1) 2^k = O(n) \quad (2.69)$$

The detailed proof is omitted due to space constraints. Hence the cumulative path length of the dummy elements to their final positions in the array is bounded by  $O(n)$ .  $\square$

We can finally conclude that there are  $O(n)$  comparisons of types 2,3 and hence the dummy elements contribute only  $O(n)$  comparisons overall.



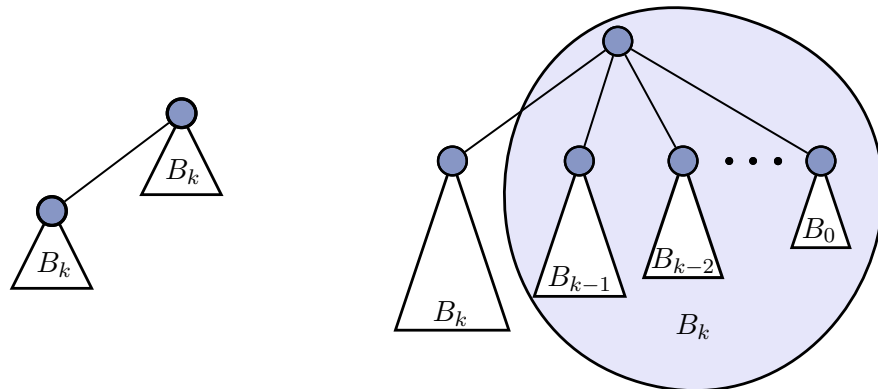
## Chapter 3

# Binomial Heapsort analysis

Although often not a method of choice for sorting, it will be shown that *Binomial Heapsort* [Vui78] algorithm preserves randomness during its runtime which makes it comparatively easy to analyze. The analysis will be carried out up to constants, as with the *Binary HeapSort*.

To recap, a binomial heap is a forest of rooted heaps (trees), ordered by their sizes and their roots stored in the *Root list*. There is only one tree for each size in the *Root list*. For every tree  $T$ ,  $|T| = 2^k$  for some  $k$  and a tree of size  $2^k$ ,  $k > 0$  is a join of two trees  $T_1, T_2$  of size  $2^{k-1}$ , with the root of  $T$  being the larger of the roots of  $T_1, T_2$ . We can also view a *Binomial heap* of size  $2^k$  as a rooted tree, with the root having  $k$  children, each being a root of a heap of size  $2^i$ ,  $0 \leq i \leq k-1$ .

Let us call the entire structure a *priority queue/Binomial heap* and a particular tree of the size of  $2^k$  for some  $k$  a *heap*.



**Figure 3.1:** An example of a binomial heap of size  $2^{k+1}$ .

Similarly to the *Binary Heapsort*, the sorting procedure consists of two phases: building the *priority queue* and popping the maximum element out of the *priority queue* until it is empty.

At the heart of the algorithm is the merge procedure, which takes two *Root lists* and merges them into another *Root list*. The procedure is very similar to binary addition of numbers.

An auxiliary *Add* procedure to deal with certain special cases of merging described below:

```

1: function ADD( $L, node, carry$ )  $\triangleright$  Merges the heaps  $node$  and the  $carry$  in case both
   are of the same size. When both are of different size, adds them to the list  $L$ 
2:   if  $carry = null$  then
3:      $L.add(node)$ 
4:     return 0
5:   else
6:     if  $carry.size = node.size$  then
7:       return Merge( $node, carry$ )
8:     else
9:        $L.add(node)$ 
10:       $L.add(carry)$ 
11:      return null

```

A *Merge* procedure to merge two heaps of the same size:

```

1: function MERGE( $L, node1, node2$ )  $\triangleright$  Merges the heaps  $node1$  and  $node2$  into a heap
   of twice the size
2:   if  $node1.value > node2.value$  then
3:      $node1.children.add(node2)$ 
4:     return  $node1$ 
5:   else
6:      $node2.children.add(node1)$ 
7:     return  $node2$ 

```

Finally another *Merge* procedure for merging two *Root lists* lists together:

```

1: function MERGE( $L_1, L_2$ )  $\triangleright$  Merges the root lists  $L_1$  and  $L_2$  to produce the list  $L_3$ 
   size
2:    $L_3 \leftarrow \{\}$ 
3:    $carry \leftarrow null$ 
4:    $t_1 \leftarrow L_1[0], t_2 \leftarrow L_2[0]$ 
5:   while  $t_1 \neq null$  and  $t_2 \neq null$  do
6:     if  $t_1.size = t_2.size$  then

```

```

7:         if carry  $\neq$  null then
8:             L3.add(carry)
9:             carry  $\leftarrow$  Merge(t1, t2)
10:            t1  $\leftarrow$  next(t1), t2  $\leftarrow$  next(t2)
11:        else
12:            if carry  $\neq$  null then
13:                if t1.size < t2.size then
14:                    carry  $\leftarrow$  Add(L3, carry, t1)
15:                    t1  $\leftarrow$  next(t1)
16:                else
17:                    carry  $\leftarrow$  Add(L3, carry, t2)
18:                    t2  $\leftarrow$  next(t2)
19:            while t1  $\neq$  null do
20:                carry  $\leftarrow$  Add(L3, carry, t1)
21:                t1  $\leftarrow$  next(t1)
22:            while t2  $\neq$  null do
23:                carry  $\leftarrow$  Add(L3, carry, t2)
24:                t2  $\leftarrow$  next(t2)
25:        return L3

```

When inserting an element  $x$  to the *priority queue*, we create a heap of size 1 containing just the element  $x$  and merge it with the *Root list*.

To remove the maximal element from the *priority queue*, we firstly need to find it in the *Root list*. Afterward, we merge every child of the maximal element back to the *Root list*. The details of all the procedures can be found in [Vui78].

### 3.1 Randomness preservation

The first thing we would like to show is that the *Binomial Heapsort* preserves randomness of the *priority queue* during the execution. Just as was done for the *Heap sort*, suppose that the input array is shuffled and that the distribution of permutations of the input elements is uniformly random.

Let us investigate the probability of a particular *priority queue* occurring.

*Lemma 17.* Let  $n/2 < 2^k \leq n$ ,  $B_n = \binom{n}{2^k} B_{2^k} B_{n-2^k}$  for  $n > 1$  and  $B_1 = 1$ . Then the number of possible *priority queues* on  $n$  elements is  $B_n$ .

*Proof.* The statement of the lemma follows from the fact that the resulting *priority queue* is uniformly random, that is any subset of elements is equally likely to appear in a heap that has its root in the *Root list*. In particular, the largest heap containing  $2^k$  elements is also uniformly random.  $\square$

*Definition 12.* A uniformly random *Binomial heap* of size  $n$  is such that each configuration of the elements obeying the heap order is equally likely, that is, has probability to occur of  $1/B_n$ .

The problem of randomness preservation of classical *Heapsort* was indicated by the fact that the number  $H_n$  (the number of possible *Binary heaps* on  $n$  elements) was not divisible by  $H_{n-1}$ . Hence after a deletion of an element from a uniformly random heap of  $n$  elements, the heap could not possibly stay uniformly random.

However with *Binomial heaps* there is a hope to show randomness preservation:

*Lemma 18.* Let  $B_n$  be defined as above, then it holds that  $B_n/B_{n-1} = n$  for odd  $n$  and  $B_n/B_{n-1} = n/2$  for even  $n$ .

*Proof.* We can show this fact by induction. There are 2 cases to consider:  $n = 2^m$  and  $n \neq 2^m$  for any  $m$ . In the first case  $B_n/B_{n-1} = \binom{n}{n/2} B_{n/2} B_{n/2} / \binom{n-1}{n/2} B_{n-1-n/2} B_{n/2} = n(n-1-n/2)B_{n/2}/(n/2)B_{n-1-n/2} = n/2$  by induction hypothesis. With a similar reasoning one can demonstrate that the second case holds. The base case of induction holds clearly as  $B_2 = 1, B_1 = 1$ .  $\square$

Using two lemmas above, show that after the *BuildHeap* procedure we obtain a uniformly random *heap*. Suppose there are two uniformly random *heaps*  $T_1, T_2$  of the same size. Fix the set of the elements contained in  $T_1 \cap T_2$  to be  $S$ .

*Lemma 19.* After merging  $T_1$  and  $T_2$ , the resulting *heap*  $T$  is uniformly random on the set  $S$ .

*Proof.* Probability of a particular *heap*  $join(T_1, T_2)$  is  $\binom{S}{|T_1|} \binom{S-|T_1|}{|T_2|} B_{|T_1|} B_{|T_2|}$  by construction, which is the same as probability of a particular 2-tuple  $(T_1, T_2)$ . As the set  $S$  was chosen arbitrarily, the argument works for any set  $S$  of an appropriate size.  $\square$

*Lemma 20.* After the *BuildHeap* procedure, the *Binomial Heap* is uniformly random, in a sense that, any *Binomial heap* configuration of the input elements is equally likely.

One of the main lemmas of the section follows immediately:

*Proof.* We can show the fact inductively, adding input elements one by one. Clearly, when we add the first element, the heap is uniformly random, as the first element in the input array is uniformly random. Let us suppose that we have added  $p$  elements, and we are adding  $p + 1$ st and so far any *Binomial heap* configuration on  $p$  elements is equally likely.

Let the *priority queue* on  $p$  elements be  $T_p$ , then the probability of a particular tuple  $(x_{p+1}, T_p)$  is the same as the probability  $\text{merge}(T_p, \{x_{p+1}\})$  in case the *Root list* of  $T_p$  does not contain a heap of size 1. In case it does contain, by the lemma above, *join* of two heaps having size a power of 2 is also uniformly random. Now we have an “overflow” heap of size 2. Clearly, we can extend the argument in case there is a heap of size 2, 4,  $\dots$  as well. Hence, after merging an element  $x_{p+1}$ , the resulting *priority queue* is uniformly random.  $\square$

*Corollary 2.* Every fixed subset of heaps is also uniformly random, in a sense that, every possible distribution of elements within the heaps is equally likely. By construction, every subheap of a heap of size  $2^s$  is also uniformly random.

So far we have shown that when the *priority queue* is built from a uniformly permuted array, its distribution is also uniformly random. What is left to show is that after a deletion of a maximum element in the *Root list*, the *priority queue* stays uniformly random.

*Lemma 21.* After deletion of a maximal element in a uniformly random *Binomial heap*, the resulting *priority queue* is also uniformly random.

*Proof.* The proof of this lemma is slightly more involved than the previous one, as the position of maximal element in a uniformly random *priority queue*’s *Root list* is not deterministic. Here is an outline of the proof: we will condition on the event that the maximal element in the *Root list* is in a particular heap. Then we will show that conditioned on this event, the resulting *priority queue* is uniformly random. In the end we will use the law of total probability to compute the probability of a particular *priority queue* occurring.

*Observation 11.* Define the set  $B_n$  as the set of *Binomial heaps* with  $n$  elements. Let the set  $B_n^p$  be the set of all the *priority queues* of size  $n$  that have a maximal element in the heap of size  $2^p$ . Then the map  $\text{PopMax} : B_n^p \rightarrow B_{n-1}$  is surjective in a sense that for every heap in  $B_{n-1}$  there exists a heap in  $B_n^p$  such that a *PopMax* transforms the later heap into the former. Define the event  $E_n^p$  indicating that the maximal element is in the heap of size  $2^p$  for a *priority queue* of size  $n$ .

*Observation 12.* For all the *priority queues*  $B \in B_n$ , the sizes of the preimages under the map *PopMax* are equal.

The observation can be shown to be true by noticing that *Binomial heaps* are isomorphic under relabelling. The two observations above indicate that the probability of a particular *Binomial heap*  $B$  after the *PopMax* operation is independent of  $B$ , and hence any  $B$  has the same probability  $(1/B_{n-1})$  of occurring.

To finish the proof, the events  $E_n^p$  for a fixed  $n$  and different  $p$  form a disjoint partition of the probability space, and hence we can apply the law of the total probability. The event  $E_B$  indicating a particular *priority queue* occurring after a *PopMax* operation, has the probability

$$Pr_{E_B} = \sum_p Pr(E_B|E_n^p)P(E_n^p) = \sum_p P(E_n^p)/B_{n-1} = 1/B_{n-1} \quad (3.1)$$

as

$$\sum_p P(E_n^p) = 1 \quad (3.2)$$

□

## 3.2 Number of comparisons

Once again, we would like to count how many comparisons does the sorting algorithm cause between a contiguous range of elements of size  $r$ . Let us argue that the number of comparisons during *BuildHeap* phase is linear in terms of the  $r$ .

*Lemma 22.* (Bounding number of comparisons during the *BuildHeap* phase) The number of comparisons between the elements of a range of size  $r$  during the *BuildHeap* phase is at most  $r$ .

*Proof.* We just have to notice that when two **red** elements are compared, their trees are merged and only the larger of the roots is “available” for further comparisons. Hence with every comparison of **red** elements, we have 1 less elements which can cause further comparisons. □

Consider a moment, when the first **red** element is being popped. We argue that before this moment, there are at most  $r$  comparisons between the **red** elements.

*Lemma 23.* If two **red** elements are compared, one of them stays the ancestor of the other until the moment they are unmerged (the ancestor is deleted). This can only happen when a **red** element is popped from the *priority queue*.

Hence we only need to consider the situation when the range  $X_r$  is the  $r$  largest elements of in the *priority queue*, as up to this event, there can be only  $r$  comparisons between the **red** elements.

So from now on we assume that there are  $n$  elements in the *priority queue* and our range  $X_r$  is the  $r$  largest elements left. Note that this is where we need randomness preservation property, as we can guarantee that at any point in the algorithm, the *priority queue* has uniform distribution.

*Theorem 2.* The number of comparisons between the **red** elements, caused by searching the maximum element in the *priority queue* is at most  $2 \ln(2)r \log(r)$ .

To establish the result above, let us look at the probabilities of particular *priority queue* configurations which cause comparisons between the **red** elements.

Clearly, if there are  $k$  **red** elements in the *Root list*, there would be at most  $k - 1$  comparisons between the **red** elements when searching the maximum element of the *Root list*.

The probability that  $k$ -th smallest **red** element is a root of a heap of size  $2^s$  is  $\binom{k-1}{2^s-1} / \binom{n}{2^s}$ . Summing the probability for all **red** elements, we have

$$\sum_{n-r+1 \leq k \leq n} \binom{k-1}{2^s-1} / \binom{n}{2^s} = 1 - \binom{n-r}{2^s} / \binom{n}{2^s} \quad (3.3)$$

*Lemma 24.* The probability that any **red** element is the root of a heap of size  $2^s$  is

$$1 - \binom{n-r}{2^s} / \binom{n}{2^s} \quad (3.4)$$

A direct result of the lemma is that the expected number of comparisons while searching for the maximal element:

*Lemma 25.* Let the binary expansion of  $n$  be  $\sum_{s_i} 2^{s_i}$ . Then the expected number of **red** elements that are in the *Root list* is

$$\sum_{s_i} 1 - \binom{n-r}{2^{s_i}} / \binom{n}{2^{s_i}} \quad (3.5)$$

such that  $2^{s_i}$  occurs in the binary expansion of  $n$ .

Notice that, after each *Pop* operation, both  $n$  and  $r$  decrease by 1, as the maximal element in the *Root list* is **red** and is being popped. To get the expected overall number of comparisons during the phase, we can sum up the expected numbers of comparisons

while decreasing  $n$ . The only difficulty is that the expression depends on the binary expansion of  $n$ .

Let us look at occurrence of  $2^s$  in the binary expansion of  $n$ , while decreasing  $n$ . It is not hard to realize that  $2^s$  occurs in the binary expansion of  $n$  in blockwise fashion: let  $n = 2^m - 1$ , then  $2^s$  will occur in the binary expansion of  $n, n-1, \dots, n-r+1$ , not occur in the binary expansion of  $n-r, n-r-1, \dots, n-2r+1$  and so on.

Let the initial number of elements in the *priority queue* be  $n_0$ . For the sake of simplifying the notation, denote  $\oplus$  the *xor* operation and  $n \oplus 2^s < n$  the case that  $2^s$  is present in the binary expansion of  $n$ . Summing the quantity in lemma above for all  $n$  such that  $n_0 - r + 1 \leq n \leq n_0$ , as for smaller  $n$ , no **red** elements are left in the heap (and hence no more comparisons are created):

$$\sum_{\substack{0 \leq k \leq r-1 \\ (n-k) \oplus 2^s < n-k}} 1 - \binom{n-r}{2^s} / \binom{n-k}{2^s} \quad (3.6)$$

It turns out that analyzing the asymptotics of the expression is hard while preserving the right constants. One could upper bound the expression by the following expression:

$$\sum_{0 \leq k \leq r-1} 1 - \binom{n-r}{2^s} / \binom{n-k}{2^s} \quad (3.7)$$

However experiments show that this approximation is approximately a 2-competitive upper bound. Instead, we rely on other means to establish the asymptotic behavior of the number of comparisons.

What we really need to count is the expected number of heaps that contain at least 1 **red** element. We can see the heaps of sizes  $2^s$  for some  $s$  as buckets where we put **red** elements.

*Observation 13.* At any moment, the number of elements in a heap of size  $2^s$  is larger than the number of elements in all the smaller heaps combined.

*Proof.* This is rather easy to see, as the number of elements in heaps is determined by the binary expansion of  $n$ . □

The next corollary should give intuition of why we would expect  $O(\log(r))$  heaps to contain at least 1 **red** element and hence the number of comparisons between the **red** elements is  $O(\log(r))$  during finding the maximal **red** element.



*Corollary 3.* If  $n = 2^k - 1$ , we expect about  $1/2$  of the **red** elements to be in the largest heap and the other  $1/4$  to be in the next largest heap, and so on. For  $n$  of a different form, even larger of fractions of elements occur in larger heaps. Hence we expect exponential decay in the number of elements “left to put” in the smaller heaps.

Consider the following process: number the heaps from largest to smallest starting from 1. We will put some number of the **red** elements to the heaps in this order. Let  $X_t$  be the number of **red** elements left (of initial size  $r$ ) before we put a number of **red** elements to the  $t$ -th heap.

*Definition 13.* Let  $T = \min\{t \in \mathbb{N}_0 | X_t = 0\}$

Clearly, the expected number of heaps that contain **red** elements is  $\leq T$ , as there could be a heap that contains no **red** elements between the heaps that have at least 1 **red** element.

The following theorem, also known as a multiplicative drift lemma, establishes behavior of  $T$  [DJW12]:

*Theorem 3.* (Multiplicative Drift lemma) Let  $\{X_t\}$  be a sequence of non-negative integer random variables. Assume that there is a  $\delta > 0$  such that

$$\forall t \in \mathbb{N}_0 : E(X_t | X_{t-1} = x) \leq (1 - \delta)x \quad (3.8)$$

then  $T = \min\{t \in \mathbb{N}_0 | X_t = 0\}$  satisfies

$$E(T) \leq (1/\delta)(\ln(X_0) + 1) \quad (3.9)$$

Let us show that our process with  $\delta = 1/2$  satisfies the requirements of the theorem:

*Proof.* As was shown before, the distribution of elements in the *Binomial Heap* is uniform, that is every subset of elements is equally likely to appear in a heap of size  $2^s$ . Or more generally, for a collection of heaps of appropriate sizes  $s_1, s_2, \dots, s_k$ , any set tuple  $(S_1, S_2, \dots, S_k)$  (where  $|S_i| = s_i$ ) is equally likely to appear as the corresponding sets of the heaps.

Now, for a single element, consider the probability  $p$  that this element occurs in the heap of size  $2^s$  (which is the largest heap) and not in the heaps of smaller size. It is not hard to see that  $p > 1/2$ , as the number of elements in the heap of size  $2^s$  is always larger than the combined number of elements in all the smaller heaps (as  $2^s > \sum_{0 \leq i \leq s-1} 2^i$ ).

Let  $H_t$  be the expected number of **red** elements in  $t$ -th heap. Then  $X_{t-1} - H_{t-1} = X_t$  and

$$E(X_t | X_{t-1} = x) = E(X_{t-1} - H_{t-1} | X_{t-1} = x) = \quad (3.10)$$

$$x - E(H_{t-1} | X_{t-1} = x) \leq x - x/2 = x/2 \quad (3.11)$$

As  $E(H_{t-1} | X_{t-1} = x) \geq xp = x/2$ . Hence the random process indeed satisfies the requirements of the theorem and

$$E(T) \leq (1/\delta)(\ln(X_0) + 1) = 2(\ln(r) + 1) \quad (3.12)$$

□

The expectation of the total number of comparisons during the *SearchMax* phase is then  $\leq 2 \ln(r!) + 2r = 2r \ln(r) + O(r)$ . From the proof of the Multiplicative Drift theorem it seems that the constant is rather tight. With a more careful analysis, we can show that the number of elements in the *Root list* that we visit before we meet the maximal element is expected to be  $O(1)$ .

Let the  $p_i$  be the probability that the maximal element is in  $i$ -th heap, then expected number of elements we visit in the *Root list* before we meet the maximal element is:

$$\sum_{1 \leq i \leq k} i p_i \prod_{1 \leq j < i} (1 - p_j) \quad (3.13)$$

The following inequalities help to establish the bound

$$q_1 q_2 \dots q_k \leq \sqrt[k]{q_1 q_2 \dots q_k} \leq (q_1 + q_2 + \dots + q_k)/k \quad (3.14)$$

for  $0 \leq q_i \leq 1$ . The first inequality follows from the fact that  $q_i \leq 1$  and the second is the standard arithmetic mean and geometric mean.

Applying the inequality to 3.13 leads to

$$\sum_{1 \leq i \leq k} i p_i \prod_{1 \leq j < i} (1 - p_j) \leq \sum_{1 \leq i \leq k} i p_i \left[ \sum_{1 \leq j < i} (1 - p_j) / (i - 1) \right] = \sum_{1 \leq i \leq k} i p_i \quad (3.15)$$

The equality follows from the fact that  $\sum_{1 \leq i \leq k} p_i = 1$ . It is easy to see that the maximum of the sum is attained when the sequence  $(p_k, p_{k-1}, \dots, p_1)$  is lexicographically largest. However the construction of the *Binomial Heap* guarantees that  $p_i \geq 2p_{i+1}$  for all  $j < k$ ,

as  $i$ -th heap is at least twice as large as the  $i + 1$ -st. And hence, the sum  $\sum_{1 \leq i \leq k} p_i = 1$  is at most  $\sum_{1 \leq i \leq \infty} i/2^i = 2$ . Note that this observation unfortunately doesn't help to find the maximal element.

Another possible improvement would be to put all the elements in the *Root list* to another *Priority queue*. This does reduce the time to find the maximal element to  $O(\log(\log(n)))$  when the number of **red** elements in is  $O(n)$ , however it is hard to argue how many comparisons between the **red** elements this causes when  $r$  (the number of **red** elements) is small, say  $O(\log(n))$ .

With the exact same technique, we can show that during the *PopMax* phase of the *Binomial Heapsort* the number of comparisons of **red** elements is also  $O(r \log r)$ . Once the maximal element in the *Root list* has been found, the heap of size  $2^s$  is split into heaps of sizes  $1, 2, \dots, 2^{s-1}$ , which are remerged with the heaps that are currently in the *Root list*. By the randomness preservation argument, the distribution of the elements in the heaps is still uniform.

*Observation 14.* When a comparison of two **red** elements happens, their corresponding heaps are merged and there is 1 less **red** element in the *Root list*.

Intuitively then, the number of comparisons that happens when *PopMax* occurs is the difference of the number of **red** elements in the *Root lists* that are being merged (the list of children of maximal element and the original *Root list*).

*Lemma 26.* Denote the expected number of **red** elements in the *Root list* of a *priority queue* of size  $n$  and having  $r$  **red** elements overall as  $R_n^r$ . Then the expected number of comparisons during a single *PopMax* call is at most

$$2R_n^r - R_{n-1}^{r-1} \quad (3.16)$$

*Proof.* We have shown that  $R_n^r \leq 2 \ln(r) + 1$ . We can apply the very same technique to show that the expected number of **red** elements that are immediate children of the maximal element (those that will be merged back to the *Root list*) is at most  $R_n^r$ . The statement of the lemma reflects the previous observation.  $\square$

Summing up the terms, we have

$$\sum_{0 \leq k \leq r-1} 2R_{n-k}^{r-k} - R_{n-1-k}^{r-1-k} = R_n^r + \sum_{1 \leq k \leq r-1} R_{n-k}^{r-k} \leq 2 \ln(r) + 2 \sum_{1 \leq k \leq r-1} \ln(k) \quad (3.17)$$

which is at most  $2 \ln(r!) = 2r \ln(r) + O(r)$ . It follows that during the *PopMax* phase there are at most  $2r \ln(r) + O(r)$  comparisons of the **red** elements, and hence the total number of comparisons during the *Binomial Heapsort* can be bounded by  $4r \ln(r) + O(r)$ .

The experiments show that the bound is not tight and real constant is around  $2\ln(2)$ , however it is not clear how to change the analysis to reduce the constant, particularly during the *FindMax* stage of the algorithm.

## Chapter 4

# Experiments

### 4.1 Binary heapsort

Experiments support the analysis of the modified version of the *Binary Heapsort*. What is interesting is that even without the modifications proposed in this thesis, based on experiments, it seems that the number of comparisons is  $r \log(r) + O(r)$ , that is the right constant for the term  $r \log(r)$  is 1.

### 4.2 Binomial heapsort

For the *Binomial heapsort*, the experiments suggest that the right constant is  $2 \ln(2)$  instead of  $4 \ln(2)$ , as was demonstrated in the analysis. Experiments show that the algorithm spends only about half the predicted time in both *FindMax* and *DeleteMax* phases, which suggests that the constant derived in the Multiplicative Drift lemma [DJW12] is not tight. We can also conclude from experiments, that the number of the red elements in the *Root list* is tightly concentrated around  $\log(r)$ ,



## Chapter 5

# Conclusion

In this thesis we have conducted analysis of *Binary Heapsort* algorithm as when used to sort a collection of strings. Proposed modifications to the algorithm provably guarantee that the total number of comparisons is small (although not optimal in the asymptotic sense, see the dependence on the prefix tree of the input strings). Unfortunately, modifications require additional linear space, which leads to a non-in-place algorithm. However we believe that the analysis presented is useful to understanding the nature of *Heapsort* algorithm.

For the *Binomial Heapsort* we were not able to show a tight constant factor for the number of comparisons of **red** elements, however we were able to demonstrate that during execution of the algorithm, random structure is preserved, which leads to a simpler and a more natural analysis than in the case of *Binary Heapsort*.

Experiments conducted demonstrate that the constant factors in the Multiplicative Drift lemma are not tight enough, which is an interesting research topic on its own. Without the use of the lemma however, the exact expressions counting the number of comparisons are hard to analyze in the asymptotic sense.





## Appendix A

# Appendix

*Definition 14.* (Rearrangement inequality)

Let  $\{a_i | 1 \leq i \leq n\}$  be an increasing sequence and  $\{b_i | 1 \leq i \leq n\}$  be decreasing. Then it holds that

$$\sum_i a_i b_i \leq \frac{(\sum_i a_i)(\sum_i b_i)}{n}$$

*Theorem 4.* (Euler-Maclaurin summation formula, first form) Let  $f(x)$  be a function defined on the interval  $(1, \infty]$  and suppose that the derivatives  $f^{(i)}(x)$  exist for  $1 \leq i \leq 2m$ , where  $m$  is a fixed constant. Then

$$\sum_{1 \leq k \leq N} f(k) = \int_1^N f(x) dx + \frac{(f(N) + f(1))}{2} + \sum_{1 \leq k \leq m} \frac{B_{2k}}{(2k)!} (f^{(2k-1)}(N) - f^{(2k-1)}(1)) + R_m$$

where  $R_m$  is a remainder s.t.  $R_m = - \int_1^N B_{2m} \frac{f^{(2m)}(x)}{(2m)!} \{1-x\} dx$  and  $B_i$  is a Bernoulli number

*Theorem 5.* (Euler-Maclaurin summation formula, second form) Let  $f(x)$  be a function defined on the interval  $(1, \infty]$  and suppose that the derivatives  $f^{(i)}(x)$  exist and are absolutely integrable for  $1 \leq i \leq 2m$ , where  $m$  is a fixed constant. Then

$$\sum_{1 \leq k \leq N} f(k) = \int_1^N f(x) dx + \frac{1}{2} f(N) + C_f + \sum_{1 \leq k \leq m} \frac{B_{2k}}{(2k)!} f^{(2k-1)}(N) + R_m$$

where  $C_f$  is a constant associated with the function  $f(x)$  and  $R_{2m}$  is a remainder term satisfying  $|R_{2m}| = O\left(\int_N^\infty |f^{(2m)}(x)| dx\right)$

*Lemma 27.* (Asymptotics of  $\sum_{r>r'>0} r' \log r'$ )

*Proof.* Fix  $m = 2$ , then using Euler-Maclaurin formula (first form), we have

$$R_m = \int_{r-1}^{\infty} \frac{B_4}{4!} \frac{2\{1-x\}}{x^3} dx < \int_{r-1}^{\infty} \frac{B_4}{4!} \frac{2}{x^3} dx < \frac{1}{(r-1)^2}$$

Integrating  $\int_1^{r-1} x \log(x) dx$  gives

$$\frac{x^2}{2} \log(x) \Big|_1^{r-1} - \int_1^{r-1} \frac{r^2}{2} \frac{1}{r} dx = \frac{(r-1)^2}{2} \log(r-1) - \frac{(r-1)^2}{4} + \frac{1}{4}$$

Hence,

$$\begin{aligned} \sum_{r>r'>0} r' \log r' &= \frac{(r-1)^2}{2} \log(r-1) - \frac{(r-1)^2}{4} + \frac{1}{4} + \frac{(r-1) \log(r-1)}{2} + \frac{B_2}{2!} \log(r-1) \\ &\quad + \frac{B_4}{4!} \left( \frac{1}{(r-1)^2} - 1 \right) + R_m, \end{aligned}$$

We can easily show that,

$\frac{1}{r} \leq \log(r) - \log(r-1)$  and hence,  $\log(r-1) \leq \log(r) - \frac{1}{r}$  and we can upper bound the expression above by

$$\frac{(r-1)^2}{2} \log(r) - \frac{(r-1)^2}{4} + \frac{(r-1) \log(r)}{2}$$

□

*Lemma 28.* (Approximation of a sum by a definite integral)

For an increasing and integrable on the summation domain function  $f$ , it holds that:

$$\int_{s=a-1}^b f(s) ds \leq \sum_{i=a}^b f(i) \leq \int_{s=a}^{b+1} f(s) ds \quad (\text{A.1})$$

Similarly, for a decreasing function  $f$  it holds that:

$$\int_{s=a}^{b+1} f(s) ds \leq \sum_{i=a}^b f(i) \leq \int_{s=a-1}^b f(s) ds \quad (\text{A.2})$$

# Bibliography

- [BS97] Jon Louis Bentley and Robert Sedgewick. Fast algorithms for sorting and searching strings. In *SODA*, pages 360–369, 1997.
- [CLRS09] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms (3. ed.)*. MIT Press, 2009.
- [DJW12] Benjamin Doerr, Daniel Johannsen, and Carola Winzen. Multiplicative drift analysis. *Algorithmica*, 64(4):673–697, 2012.
- [Flo64] Robert W. Floyd. Algorithm 245: Treesort. *Commun. ACM*, 7(12):701, 1964.
- [Fri56] E. Friend. Sorting on electronic computer systems. *J. ACM*, 3:134–168, 1956.
- [GKP94] Ronald L. Graham, Donald E. Knuth, and Oren Patashnik. *Concrete mathematics - a foundation for computer science (2. ed.)*. Addison-Wesley, 1994.
- [Han02] Yijie Han. Deterministic sorting in  $o(n \log \log n)$  time and linear space. In *STOC*, pages 602–608, 2002.
- [HT02] Yijie Han and Mikkel Thorup. Integer sorting in  $O(n \sqrt{\log \log n})$  expected time and linear space. In *FOCS*, pages 135–144, 2002.
- [Knu73] Donald E. Knuth. *The Art of Computer Programming, Volume III: Sorting and Searching*. Addison-Wesley, 1973.
- [Sei10] Raimund Seidel. Data-specific analysis of string sorting. In *SODA*, pages 1278–1286, 2010.
- [SF96] Robert Sedgewick and Philippe Flajolet. *An introduction to the analysis of algorithms*. Addison-Wesley-Longman, 1996.
- [SS93] Russel Schaffer and Robert Sedgewick. The analysis of heapsort. *J. Algorithms*, 15(1):76–100, 1993.
- [VCFF09] Brigitte Vallée, Julien Clément, James Allen Fill, and Philippe Flajolet. The number of symbol comparisons in quicksort and quickselect. In *ICALP (1)*, pages 750–763, 2009.

- [Vui78] Jean Vuillemin. A data structure for manipulating priority queues. *Commun. ACM*, 21(4):309–315, April 1978.
- [Wik] Wikipedia. Sorting algorithm.
- [Wil64] J. W. J. Williams. Algorithm 232: Heapsort. *Communications of the ACM*, 7(6):347–348, 1964.